Distributed diagnosis system for osteoarthritis

Alessandro Magnani - 0001026336
alessandro.magnani18@studio.unibo.it

Andrea Matteucci - 0001026901
andrea.matteucci5@studio.unibo.it

Simone Montanari - 0001026332
simone.montanari14@studio.unibo.it

December 2024

Il progetto è incentrato sullo sviluppo di un sistema distribuito per supportare la diagnosi dell'osteoartrite delle ginocchia tramite l'analisi di radiografie. Il sistema consente a pazienti e medici di caricare le proprie radiografie tramite un'applicazione web, che sfrutta una rete neurale convoluzionale (CNN) pre-addestrata per fornire diagnosi automatizzate.
Il sistema è stato containerizzato utilizzando Docker, assicurando portabilità, modularità e una gestione semplificata dell'infrastruttura. Successivamente, è stato distribuito su una macchina virtuale ospitata su Google Cloud Compute Engine, permettendo l'accesso al sistema tramite l'indirizzo IP pubblico della VM. Questo approccio ha garantito una distribuzione scalabile e facilmente accessibile.

Indice

Elenco delle figure

Listati

1 Obiettivi/Requisiti

Si vuole realizzare un sistema distribuito per supportare medici e pazienti nella gestione e nell'analisi di radiografie delle ginocchia, con particolare riferimento alla diagnosi di osteoartrite. Il sistema, accessibile tramite un'applicazione web, integra una rete neurale pre-addestrata per classificare il grado di gravità dell'osteoartrite e fornire supporto alle decisioni cliniche.

Gli utenti del sistema si dividono in due categorie: medici e pazienti. I pazienti possono visualizzare le proprie radiografie e accedere alle diagnosi fornite dal sistema, mentre i medici possono caricare le radiografie dei pazienti, analizzarle, pianificare operazioni e fornire feedback sulle diagnosi generate dal sistema.

Il sistema prevede una dashboard interattiva per consultare lo storico delle diagnosi, complete di informazioni sul grado di gravità e heat map che evidenziano le aree critiche delle radiografie. È inoltre disponibile un'interfaccia calendario per la gestione di attività e operazioni pianificate, sincronizzata con un sistema di notifiche che informa pazienti e medici sugli eventi rilevanti.

Un elemento chiave del sistema è il miglioramento continuo delle prestazioni della rete neurale. Le radiografie valutate positivamente dai medici vengono utilizzate per il riaddestramento periodico del modello, garantendo una progressiva ottimizzazione delle predizioni.

Dal punto di vista tecnico, il sistema è progettato per essere scalabile e sicuro, con una gestione efficiente delle richieste lato server e un'archiviazione sicura per radiografie e diagnosi. L'obiettivo finale è quello di migliorare la precisione diagnostica, semplificare il flusso di lavoro medico e favorire un'interazione intuitiva tra utenti e tecnologia.

1.1 Scenarios

I principali gruppi di utenti sono i medici e i pazienti, ciascuno con esigenze e aspettative specifiche che il sistema deve soddisfare.

1.1.1 Medici

I medici, in particolare i chirurghi, sono il pubblico di riferimento primario per la piattaforma. La loro necessità principale è quella di *poter consultare facilmente e velocemente le diagnosi e le radiografie dei pazienti*, con una chiara visualizzazione delle aree critiche, che possano essere evidenziate tramite le heat map fornite dalla rete neurale. Il sistema deve permettere ai medici di *pianificare e gestire gli interventi chirurgici o le attività legate ai pazienti in modo semplice ed efficace*, senza dover utilizzare strumenti separati per la gestione delle radiografie e per la programmazione degli eventi.

Inoltre i medici necessitano di un sistema che gestisca in modo automatico le diagnosi, riducendo la complessità nella valutazione delle immagini. La piattaforma deve offrire anche funzionalità di notifica per avvisarli quando ci sono aggiornamenti relativi agli appuntamenti o alle diagnosi.

Personas: Dr. Marco

Dr. Marco è un chirurgo che cerca un sistema che lo colleghi direttamente ai suoi pazienti, permettendogli di visualizzare rapidamente le radiografie e le diagnosi associate. La sua priorità è ridurre il tempo speso nella gestione delle immagini e delle diagnosi, così da concentrarsi maggiormente sulla pianificazione delle operazioni. Poiché determinare se un ginocchio ha necessità di un intervento chirurgico è un processo complesso e spesso difficile da intuire, Dr. Marco desidera avere una rete neurale esperta e affidabile come supporto, per aiutarlo nelle decisioni e aumentare la precisione delle sue valutazioni.

Scenario d'uso: Dr. Marco ha bisogno di un sistema che gli consenta di accedere rapidamente alle radiografie dei suoi pazienti e di ottenere diagnosi supportate da intelligenza artificiale.

1. Dr. Marco accede alla piattaforma e si autentica.
2. Visualizza le radiografie dei pazienti e le diagnosi associate.
3. Esamina la gravità dell'osteoartrite tramite le heat map generate dalla rete neurale.
4. Prende decisioni informate sui trattamenti, con l'affidabilità dell'IA a supporto della sua esperienza clinica.

1.1.2 Pazienti

I pazienti sono il secondo gruppo di utenti del sistema. Le loro esigenze sono incentrate sulla possibilità di visualizzare le proprie radiografie e comprendere la gravità della propria condizione.

I pazienti devono poter *consultare le diagnosi in modo chiaro*, con l'utilizzo di heat map che aiutano a comprendere meglio il proprio stato di salute. Inoltre, il sistema deve inviare notifiche per aggiornamenti sui propri appuntamenti, come quando un medico pianifica un intervento chirurgico o quando ci sono cambiamenti nel piano di cura.

Personas: Laura

Laura è una paziente che ha recentemente ricevuto una diagnosi di osteoartrite e ha bisogno di monitorare l'evoluzione della sua condizione. Non ha molta esperienza con la tecnologia, quindi la piattaforma deve essere facile da usare.

Scenario d'uso:

1. Laura accede alla piattaforma con le proprie credenziali.
2. Visualizza le radiografie caricate dal suo medico.
3. Seleziona una radiografia e visualizza la diagnosi automatica, inclusa la heat map con le aree critiche evidenziate.
4. Riceve una notifica per un appuntamento pianificato dal medico, con tutti i dettagli necessari.

1.1.3 Sfide comuni e soluzioni progettuali

Le principali sfide comuni tra medici e pazienti sono la gestione di informazioni complesse e la necessità di una piattaforma che sia allo stesso tempo potente e facile da usare. I medici richiedono uno strumento avanzato per analizzare le radiografie, ma senza perdite di tempo o complicazioni, mentre i pazienti necessitano di un'interfaccia chiara e accessibile che non richieda competenze tecniche.

Il sistema deve integrare tutte le funzionalità necessarie, come la visualizzazione delle immagini, la pianificazione degli interventi e la gestione delle notifiche, in un'unica piattaforma centralizzata. Ciò ridurrà la necessità di coordinarsi tra più sistemi, come avviene attualmente per la gestione di radiografie e appuntamenti separati.

Per ottimizzare l'esperienza utente, il sistema è progettato per essere accessibile su diverse piattaforme (desktop, tablet e smartphone), garantendo che il target possa utilizzare la piattaforma senza difficoltà, indipendentemente dal dispositivo scelto.

1.1.4 Scenario d'uso

Supponiamo che i target user definiti prima abbiano necessità di usare l'applicazione. Una volta aperta, all'utente, sia esso medico o paziente, verrà chiesto di registrarsi oppure di effettuare il login se già in possesso di un account.

Durante la registrazione, il sistema chiede all'utente se desidera iscriversi come medico o come paziente. In entrambi i casi è necessario inserire i propri dati personali, ma con una differenza importante: il medico deve fornire il proprio ID di iscrizione all'albo, mentre il paziente deve selezionare uno tra i medici esistenti. Una volta completata la registrazione, viene inviata un'email di verifica che l'utente dovrà confermare per attivare l'account.

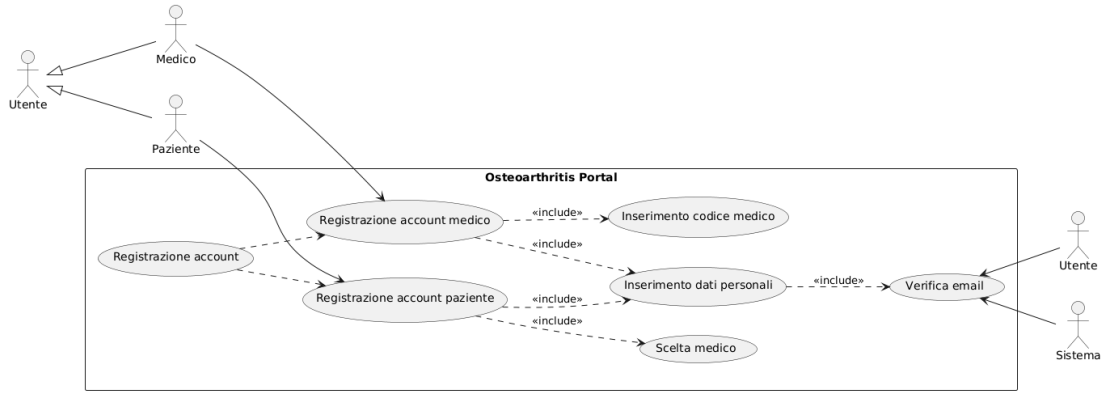

Figura 1: Diagramma casi d'uso: registrazione di un nuovo utente.

Per quanto riguarda l'accesso, l'utente deve inserire le proprie credenziali. Il sistema controlla se i dati sono corretti e se l'email è stata verificata. Se tutto è in regola, l'utente viene reindirizzato alla Home; in caso contrario, viene mostrato un messaggio di errore. Se necessario, è possibile reimpostare la password: il sistema invia un'email con un link per crearne una nuova.

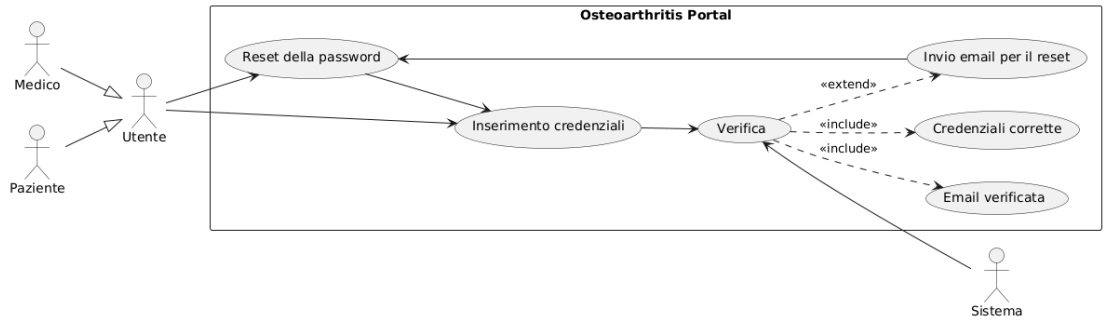

Figura 2: Diagramma casi d'uso: login.

Una volta effettuato l'accesso, dalla Home l'utente può consultare i propri dati e accedere alla sezione delle radiografie. I pazienti possono visualizzare le proprie radiografie, mentre i medici possono selezionare uno dei loro pazienti per consultare le immagini caricate. Sia medici che pazienti possono visualizzare la diagnosi associata a ciascuna radiografia.

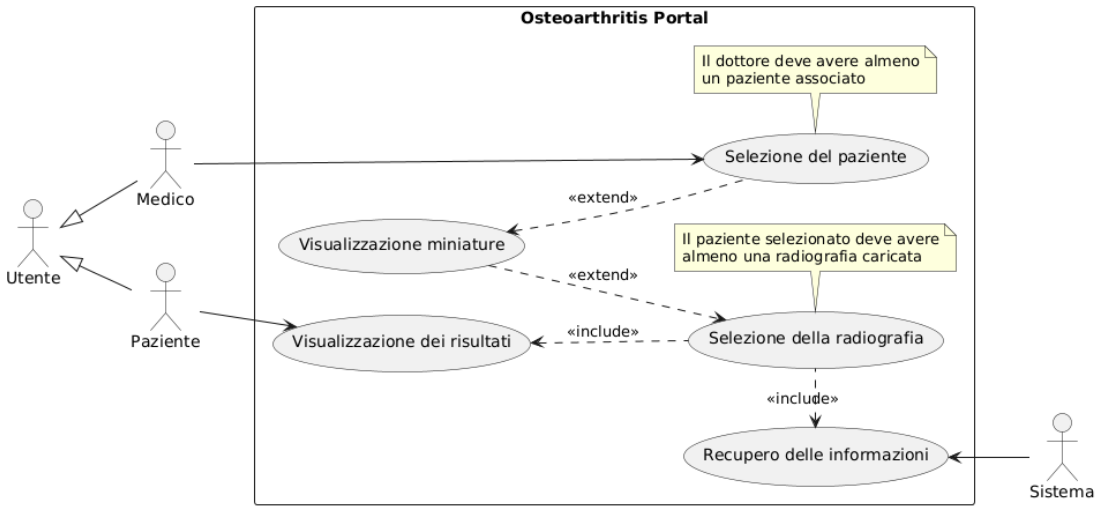

Figura 3: Diagramma casi d'uso: visualizzazione radiografie.

Solo i medici hanno la possibilità di caricare nuove radiografie, eseguire una predizione e visualizzarne i risultati, che possono anche essere commentati.

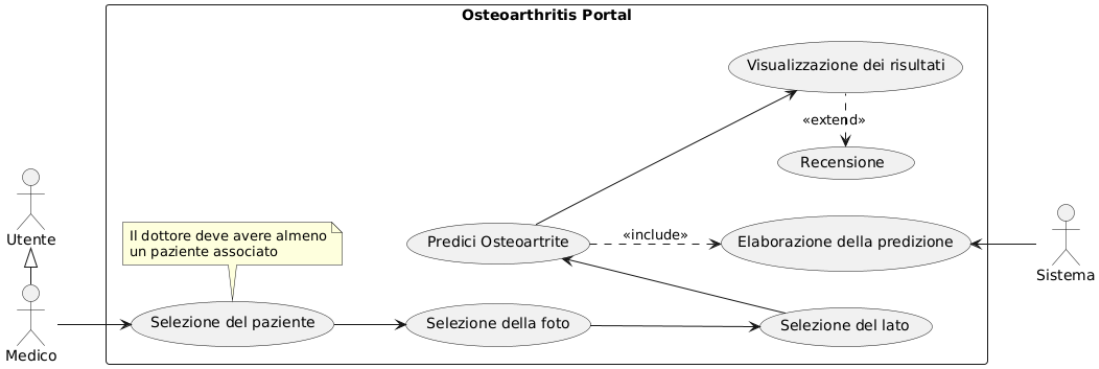

Figura 4: Diagramma casi d'uso: predizione.

Entrambi gli utenti possono accedere al calendario, dove sono visibili tutti gli impegni futuri. Oltre alle radiografie, nel calendario possono comparire anche le operazioni programmate. Solo i medici possono pianificare un intervento selezionando un paziente: una volta programmata, il sistema invia automaticamente una notifica al paziente con tutte le informazioni necessarie. Il paziente può leggere questa notifica e segnarla come letta.

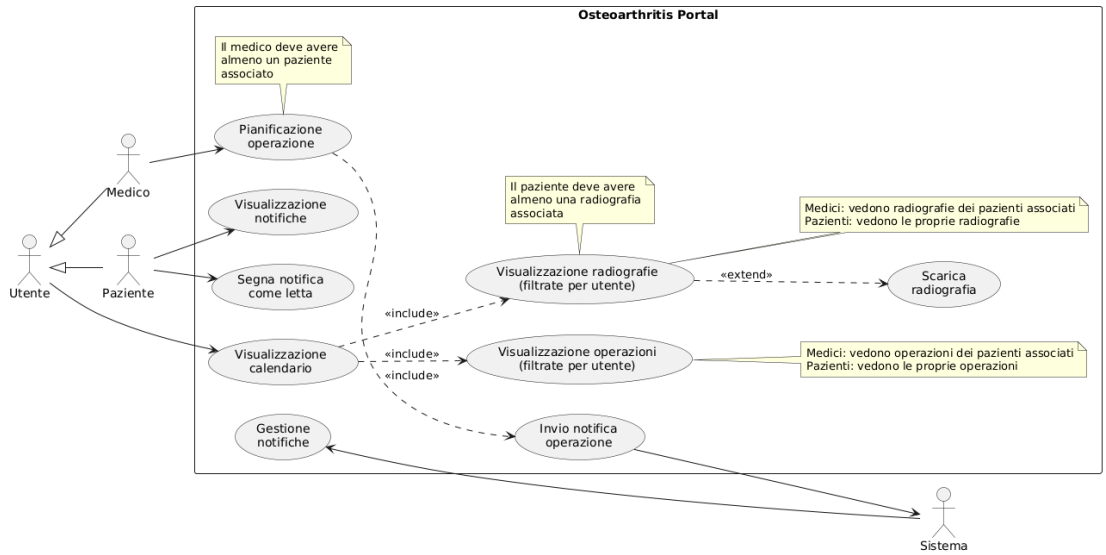

Figura 5: Diagramma casi d'uso: visualizzazione calendario e gestione notifiche.

1.2 Metodi di autovalutazione

Per valutare l'efficacia del software sviluppato, saranno utilizzati sia test automatici sia test manuali. I test automatici verificheranno principalmente le funzionalità del backend, mentre quelli manuali controlleranno la reattività e il comportamento dell'interfaccia grafica. I dettagli sui test saranno approfonditi nella sezione 5. La qualità del servizio offerto sarà verificata controllando se l'applicazione soddisfa tutti i requisiti definiti nell'analisi e se raggiunge gli obiettivi prefissati in modo efficace. Per valutare l'interfaccia utente, saranno seguiti i principi dell'euristica di Nielsen attraverso più sessioni di Cognitive Walkthrough, svolte durante le varie fasi dello sviluppo Agile. Questo metodo consente di individuare e risolvere rapidamente eventuali problemi di usabilità. L'esperto del dominio con cui collaboriamo ha partecipato a un test di usabilità, fornendo un feedback completo sulla funzionalità del sistema, sull'usabilità dell'interfaccia e sull'efficacia delle diagnosi. Il suo contributo ha supportato il miglioramento del software.

2 Analisi dei requisiti

Sono di seguito riportati i requisiti del sistema, suddivisi in requisiti funzionali e non funzionali.

2.1 Requisiti Funzionali

- **Gestione degli accessi**:
  Il sistema deve distinguere due tipologie di utenti (pazienti e medici), fornendo le seguenti funzionalità:

  - Registrazione di nuovi utenti, con differenziazione dei privilegi in base al tipo.
  - Verifica dell'indirizzo email tramite link di conferma personalizzato.
  - Gestione dei tentativi di login falliti.
  - Reset della password tramite link personalizzato, attivato al superamento del numero di tentativi falliti (6).
  - Accesso al sistema tramite login.

- **Gestione delle radiografie**
  Pazienti:
  - Possono visualizzare le proprie radiografie.

  Medici:
  - Possono caricare e visualizzare le radiografie per ogni paziente associato.

- **Gestione delle diagnosi**
  Il sistema deve integrare una rete neurale (*ResNet50*) pre-addestrata per predire il grado di osteoartrite presente nelle radiografie secondo la scala KL.
  Pazienti:
  - Possono visualizzare le proprie diagnosi.

  Medici:
  - Possono effettuare nuove diagnosi e visualizzare quelle esistenti per ogni paziente associato.

- **Gestione delle operazioni**
  Medici:
  - Possono pianificare operazioni per i pazienti associati, impostando il giorno, l'ora e la descrizione dell'intervento.

- **Gestione delle notifiche**
  Il sistema deve inviare notifiche automatiche ai pazienti quando un medico pianifica un'operazione. Le notifiche devono includere informazioni rilevanti come la data e l'ora dell'intervento.

- **Gestione del calendario delle attività**
  Il sistema deve consentire la gestione di due tipi di attività: interventi chirurgici e caricamento delle radiografie. Le attività devono essere visualizzate tramite un calendario integrato nel sistema. Pazienti:
  - Possono visualizzare gli interventi pianificati e le date in cui sono state caricate le proprie radiografie.

  Medici:
  - Possono pianificare nuove interventi chirurgici o modificare quelli esistenti.

  - Possono visualizzare tutte le operazioni e tutte le date dei caricamenti per ogni paziente associato.

- **Sistema di feedback e miglioramento continuo**
  Medici:
  - Possono valutare le diagnosi fornite dal sistema.
  - Le valutazioni influenzano, in misura controllata, i successivi processi di predizione.
  - Le radiografie che ricevono una valutazione di 5 stelle vengono inserite nel database, insieme a tutte le altre immagini utilizzate per il riaddestramento periodico della rete neurale.
  - Il riaddestramento viene effettuato in batch, migliorando la qualità delle predizioni future.

2.2 Requisiti Non Funzionali

- **Usabilità**
  - L'interfaccia utente deve essere intuitiva.
  - L'interfaccia deve essere utilizzabile anche da utenti non esperti.

- **Sicurezza**
  - Devono essere implementati limiti sui tentativi di login per prevenire attacchi di *brute-force*.
  - Tutte le credenziali devono essere protette tramite crittografia.
  - Il sistema deve verificare i token di autenticazione utilizzando meccanismi sicuri, come l'uso di *Bearer token* nei protocolli HTTP, per garantire che solo gli utenti autorizzati possano accedere alle risorse.

- **Scalabilità**
  - Deve essere possibile aggiungere nuove funzionalità senza compromettere il design esistente.

- **Manutenibilità**
  - Il sorgente dev'essere commentato e strutturato per consentire modifiche e aggiornamenti futuri.
  - Devono essere presenti *log* dettagliati per monitorare lo stato del sistema e facilitare il debugging.

2.3 Analisi e Modello del Dominio

Dato che l'applicazione consiste in un sistema Web progettato per gestire in modo integrato diagnosi mediche e attività cliniche, si intende realizzare il programma aderendo al modello dei servizi *RESTful*. Questo approccio consentirà di implementare un'interfaccia uniforme, riutilizzabile e facilmente estendibile per l'accesso alle funzionalità del sistema.

Le entità principali del dominio che si prevede di modellare saranno:

- **Client web**: costituirà i punti di accesso al sistema per medici e pazienti. Specificando opportune rotte API, i client potranno eseguire operazioni come il caricamento e la visualizzazione delle radiografie, la consultazione delle diagnosi e la gestione degli interventi chirurgici pianificati.

- **Server web**: si occuperà di esporre il servizio ai client, definendo le rotte API necessarie e implementando la logica applicativa. Sarà responsabile di interagire con un database non relazionale.

- **Rete neurale**: verrà integrata nel sistema una rete neurale convoluzionale preaddestrata (*ResNet50*), in grado di classificare il grado di osteoartrite presente nelle radiografie secondo la scala *KL*. La rete sarà progettata per essere periodicamente riaddestrata utilizzando dati di alta qualità, selezionati in base al feedback fornito dai medici.

- **Database non relazionale**: si occuperà di mantenere le informazioni relative agli utenti (profili e privilegi), alle radiografie, alle diagnosi e alle operazioni pianificate. Inoltre, fornirà tali informazioni al server su richiesta, consentendo l'esecuzione delle operazioni necessarie. Il database sarà ottimizzato per garantire sicurezza ed efficienza. La gestione dei dati sarà flessibile grazie alla struttura del database non relazionale, consentendo di scalare facilmente con l'aumentare della complessità e delle dimensioni del sistema.

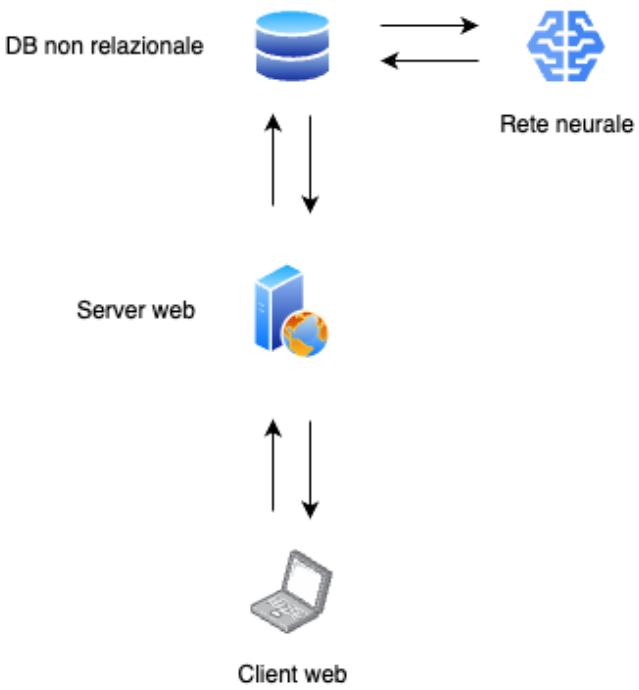

Figura 6: Entità del dominio e le loro interazioni.

3 Design

Alla luce delle informazioni raccolte durante le fasi di analisi, si è passati alla fase di progettazione, in cui sono state definite la struttura delle entità coinvolte e le loro interazioni.

3.1 Struttura

Come spiegato in precedenza il nostro dominio è costituito da quattro entità base: database non relazionale, Server, Client Web e Rete neurale.

Per garantire una chiara separazione dei compiti e una maggiore modularità, il sistema è stato suddiviso in due container distinti, uno per il backend e uno per il frontend, orchestrati attraverso Docker (2). Il container dedicato al backend gestisce tutta la logica applicativa, inclusa la comunicazione con il database non relazionale. Questo container è esposto sulla porta 5000 ed è progettato per ricevere richieste dai client ed eseguire le operazioni necessarie, come il recupero o l'elaborazione dei dati.

Dall'altro lato, il container frontend è responsabile della presentazione visiva e dell'interazione con l'utente finale. Comunica con il backend tramite API REST, inviando richieste specifiche e mostrando i risultati attraverso un'interfaccia grafica accessibile

sulla porta 8080. Entrambi i container sono collegati tramite una rete Docker di tipo bridge. Questo tipo di rete consente ai container di comunicare tra loro in modo diretto utilizzando i rispettivi nomi di servizio come endpoint. Ad esempio, il frontend può accedere ai servizi del backend tramite un URL interno, semplificando le operazioni di scambio di dati e mantenendo un alto grado di isolamento rispetto all'esterno.

Di conseguenza, la macro-struttura del nostro sistema si presenta in questo modo.

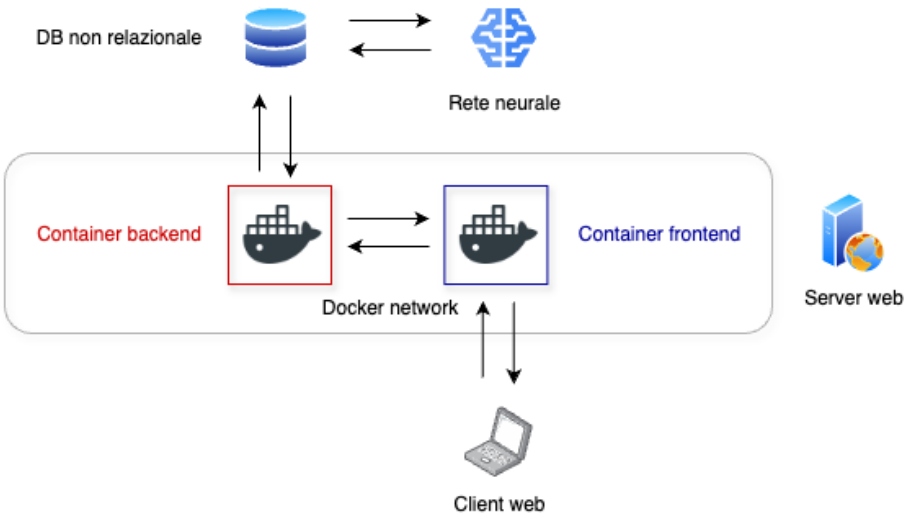

Figura 7: Struttura del sistema

3.1.1 Struttura del Database

Per prima cosa è stato definito lo schema Entity/Relationship del database, mettendo in luce quali sono le diverse entità che dovranno essere modellate e le loro relazioni.

Come si può vedere dalla figura 8, lo schema è costituito dalle seguenti entità:

- **Users**, rappresenta gli utenti iscritti all'applicazione. Un utente può essere un medico (*Doctor*) o un paziente (*Patient*) ed è identificato in modo univoco tramite l'attributo `id`.
- **Doctors**, rappresenta i medici registrati nel sistema, associati agli utenti tramite l'attributo `doctorId`.
- **Patients**, rappresenta i pazienti registrati nel sistema. Un paziente è associato a un medico attraverso l'attributo `doctorRef`.
- **Radiographs**, rappresenta le radiografie dei pazienti. Poiché le radiografie sono immagini non strutturate, sono memorizzate in Google Cloud Storage e non hanno attributi specifici, tranne un identificativo `radiographId`.

- **Predictions**, rappresenta le predizioni fatte dai medici sulle radiografie. Anche le predizioni sono memorizzate in Google Cloud Storage e sono collegate alle radiografie tramite l'attributo `predictionId`.
- **Models**, rappresenta l'architettura del modello di rete neurale preaddestrata. Anche questi sono dati non strutturati e vengono utilizzati dai medici per predire le radiografie.
- **Operations**, rappresenta le operazioni pianificate per i pazienti. Ogni operazione è associata a un paziente e un medico specifico tramite gli attributi `patientId` e `doctorId`.
- **Notifications**, rappresenta le notifiche inviate ai pazienti riguardanti le operazioni pianificate. Ogni notifica è associata a un paziente tramite l'attributo `patientId`.

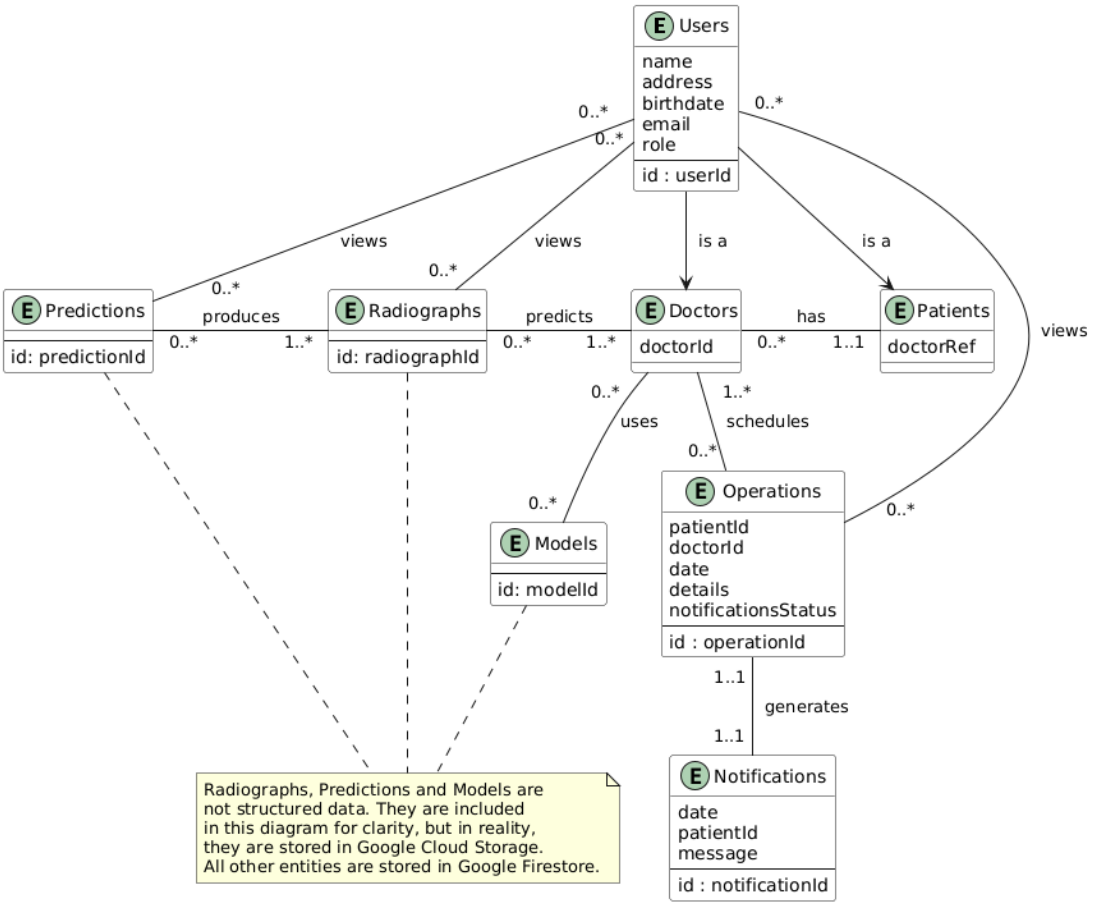

Figura 8: Schema Entity/Relationship del Database

Come già sottolineato, le radiografie (`Radiographs`), le predizioni (`Predictions`) e i pesi (`Weights`) sono dati non strutturati e vengono memorizzati separatamente in Goo-

gle Cloud Storage. Difatti queste entità non hanno attributi complessi, ma sono solo immagini o file di predizione. Questi dati sono stati inclusi nel diagramma ER per chiarezza anche se in realtà non sono parte della struttura del database Firestore (5).

Le operazioni (`Operations`) sono pianificate dai medici per i pazienti. Ogni operazione è legata a una notifica (`Notifications`), che viene inviata al paziente una volta che l'operazione è stata programmata. Ogni operazione genera una notifica, e ogni notifica è associata a un singolo paziente.

La modellazione segue il principio di separare chiaramente i dati strutturati (salvati in Firestore) dai dati non strutturati (salvati in Google Cloud Storage), per semplificare la gestione del database e migliorare le performance.

3.1.2 Struttura del sistema: moduli principali

Il progetto è stato sviluppato suddividendo il sistema in diversi moduli funzionali, garantendo una chiara separazione delle responsabilità e una gestione efficiente delle funzionalità. I moduli principali del sistema sono i seguenti:

- **Core**: rappresenta il cuore del sistema, coordina tutte le operazioni e i servizi. È responsabile di inizializzare e configurare il sistema. Coordina l'interazione tra i diversi moduli per assicurare il corretto svolgimento delle operazioni
- **Utility**: fornisce servizi di supporto per i diversi moduli del sistema, gestendo operazioni specifiche come l'invio di notifiche, la gestione dei dati, l'archiviazione e le operazioni del modello. Questo modulo garantisce un insieme di strumenti affidabili per supportare le funzionalità principali del sistema
- **Controller**: è il modulo responsabile della logica di business. Coordina le operazioni richieste dagli utenti, interfacciandosi con i servizi e le funzionalità del sistema per garantire una gestione coerente e strutturata dei processi
- **Factory**: si occupa della creazione e configurazione dei Manager, assicura che i servizi e i moduli siano inizializzati e configurati correttamente.
- **Routing**: gestisce l'instradamento delle richieste all'interno del sistema, assicura che ogni richiesta venga elaborata dal modulo appropriato. Mappa le interazioni degli utenti verso le operazioni corrispondenti, permettendo di comunicare in modo organizzato all'interno del sistema
- **Config**: si occupa della gestione della configurazione del sistema come le credenziali per accedere a Firestore e GCS o le variabili di configurazione di email e model
- **UI**: si occupa della presentazione e dell'interazione con l'utente, gestendo le diverse viste e funzionalità dell'applicazione. Consente agli utenti di accedere alle infor-

mazioni e ai servizi offerti dal sistema, garantisce un'esperienza fluida e intuitiva seguendo i principi dell'euristica di Nielsen

- **Service**: è il modulo che gestisce la comunicazione tra l'interfaccia utente e il sistema centrale. Si occupa della gestione delle richieste e delle risposte e garantisce un flusso di dati strutturato ed efficiente.

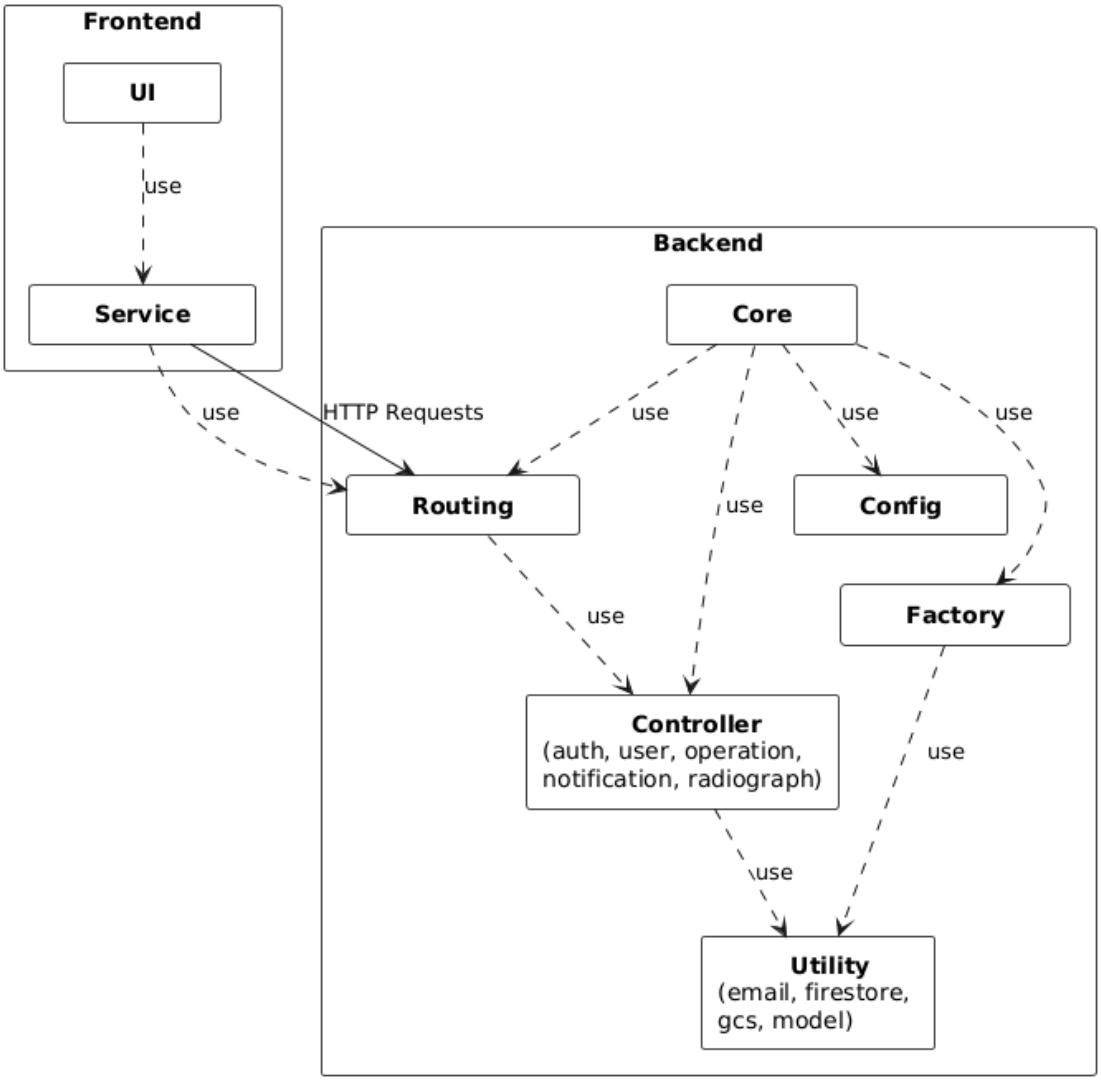

Figura 9: Diagramma delle dipendenze

Come si può vedere in Figura 9 il flusso di interazioni tra i moduli ha origine dal frontend dove i componenti dell'interfaccia utente interagiscono con il modulo dei **Servizi**. Questo servizio gestisce le chiamate HTTP verso il backend, comunicando principalmen-

te con il modulo **Routing** che espone le API attraverso delle rotte specifiche.

Nel backend il **Core** coordina tutte le interazioni tra i vari moduli. Difatti si occupa di orchestrare la creazione delle istanze delle **Utility** tramite il modulo **Factory**. Le **Utility** sono poi iniettate nei **Controllers**, i quali costituiscono il punto di contatto tra il **Routing** e la logica di business del sistema.

Il modulo **Routing** infatti riceve le richieste HTTP dal frontend e le indirizza ai **Controllers** appropriati. Il sistema è configurato attraverso il modulo **Config** che fornisce i parametri essenziali per il corretto funzionamento dell'applicazione.

Questa architettura garantisce una netta separazione delle responsabilità: ogni modulo ha un ruolo ben definito e comunica con gli altri tramite interfacce chiare e strutturate. La comunicazione tra il frontend e il backend avviene esclusivamente tramite chiamate HTTP, mantenendo un adeguato disaccoppiamento tra i due sistemi.

3.1.2.1 Modulo Core

Il modulo Core è il punto di orchestrazione del sistema, implementando un **server stateless basato su Flask**. La scelta di un'architettura stateless è stata dettata dalla necessità di garantire scalabilità e manutenibilità del sistema: ogni richiesta contiene tutte le informazioni necessarie per la sua elaborazione, eliminando la necessità di mantenere uno stato server-side e facilitando così la gestione delle risorse (3).

Il Core adotta il pattern `Factory` per la creazione e la gestione delle dipendenze, un approccio che favorisce la modularità e semplifica l'aggiunta di nuovi componenti. Questa scelta progettuale permette di centralizzare la logica di inizializzazione. Questa separazione è implementata attraverso un sistema di dependency injection, dove il Core si occupa di inizializzare e iniettare le dipendenze necessarie nei vari controllers.

Un aspetto fondamentale del design è la configurabilità del sistema. Il Core sfrutta un approccio basato su variabili d'ambiente, permettendo una facile configurazione in diversi ambienti di deployment senza necessità di modifiche al codice. La natura stateless del server, combinata con l'utilizzo di Flask, permette di ottenere un sistema altamente scalabile e performante, capace di gestire multiple richieste in modo efficiente liberando rapidamente le risorse.

3.1.2.2 Modulo Utility

Il modulo Utility è stato progettato per fornire un insieme di funzionalità di supporto essenziali per l'interazione con vari servizi esterni e per l'elaborazione dei dati. Il modulo è stato suddiviso in quattro componenti principali, ognuno specializzato in un aspetto specifico del sistema:

- `FirestoreManager`: gestisce tutte le interazioni con il database Firestore, fornendo un'interfaccia uniforme per le operazioni CRUD (Create, Read, Update, Delete) e implementando funzionalità specifiche per il dominio applicativo.
- `GCSManager`: si occupa dell'interazione con Google Cloud Storage. Le funzionalità principali sono le operazioni CRUD sui file nel bucket di storage, il caricamento e la gestione dei modelli di machine learning e delle radiografie e la generazione di URL pubblici per l'accesso ai file
- `EmailManager`: fornisce un'interfaccia semplificata per l'invio di email attraverso il servizio SMTP di Gmail.
- `ModelManager`: gestisce le operazioni relative al modello di machine learning per l'analisi delle radiografie. In particolare si occupa del preprocessing delle immagini per l'input al modello, l'esecuzione delle predizioni, la generazione di heatmap e l'elaborazione delle immagini per la visualizzazione dei risultati

Figura 10: Diagramma delle classi: modulo Utility

3.1.2.3 Modulo Controller

Il modulo Controller gestisce la logica di business e fa da intermediario tra le richieste degli utenti e i servizi del sistema. Il modulo è organizzato secondo il principio della separazione delle responsabilità, con ogni controller dedicato a uno specifico dominio

dell'applicazione. La struttura del modulo si basa su un'architettura orientata agli oggetti dove ogni controller è implementato come una classe che riceve le sue dipendenze attraverso il costruttore (dependency injection). Si è pensato a questo approccio dal momento che rende più semplice testare e manutenere il codice.

Il modulo si articola in cinque controller specializzati:

- `AuthController`: gestisce tutti gli aspetti legati all'autenticazione e all'autorizzazione degli utenti, inclusa la registrazione, il login, la verifica dell'email e il reset della password. Integra i servizi di Firebase Authentication per garantire una gestione sicura delle credenziali.
- `UserController`: si occupa della gestione degli utenti del sistema, fornisce funzionalità per la gestione dei profili, la distinzione tra pazienti e dottori e le relazioni tra questi. Implementa logiche specifiche per il recupero e l'aggiornamento dei dati utente.
- `RadiographController`: responsabile della gestione delle radiografie, incluso il caricamento, l'elaborazione e l'analisi delle immagini. Integra funzionalità di machine learning per la predizione diagnostica e la generazione di heatmap (Grad-CAM).
- `OperationController`: gestisce la pianificazione e il tracciamento delle operazioni mediche, mantiene le relazioni tra pazienti e interventi programmati.
- `NotificationController`: implementa il sistema di notifiche, gestisce l'invio e il tracciamento delle comunicazioni tra il sistema e gli utenti.

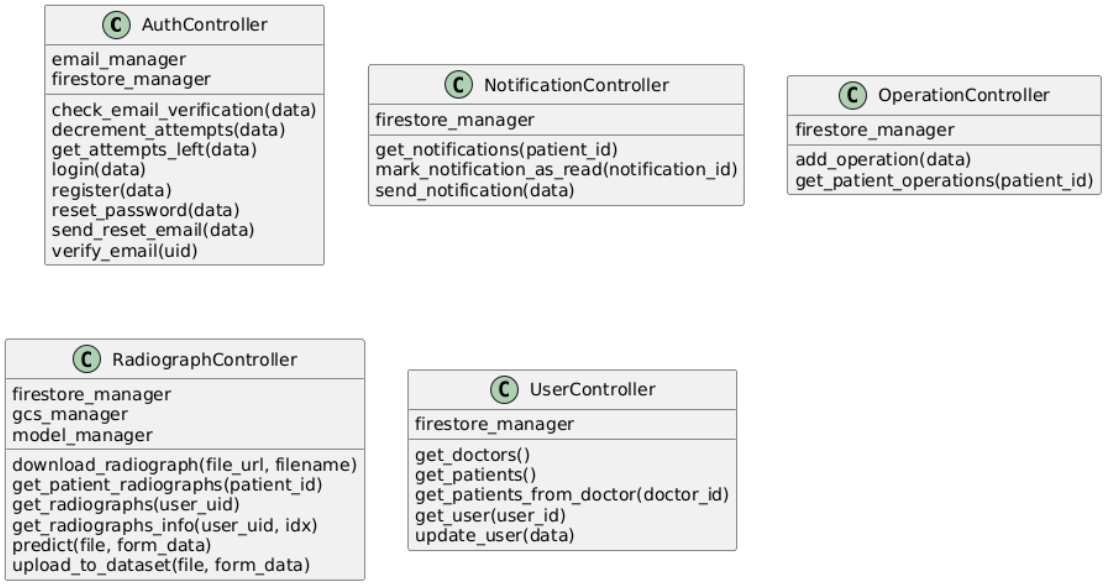

Figura 11: Diagramma delle classi: modulo Controller, ogni classe è indipendente e si occupa di un dominio differente, totale separazione delle responsabilità.

Ogni controller implementa una REST API che espone le proprie funzionalità attraverso endpoint HTTP, seguendo le best practice REST. La gestione degli errori è centralizzata: ogni controller implementa appropriate strategie di error handling e reporting.

L'architettura adottata permette una facile estensione del sistema: nuove funzionalità possono essere aggiunte implementando nuovi controller o estendendo quelli esistenti, senza impattare sul resto del sistema.

3.1.2.4 Modulo Factory

Il modulo Factory si basa sulla classe `ManagerFactory` che, seguendo il principio di Single Responsibility, ha la specifica responsabilità di creare e configurare i diversi manager del sistema. L'utilizzo del pattern Factory permette di incapsulare la logica di creazione degli oggetti, nascondendo i dettagli implementativi e fornendo un'interfaccia pulita per istanziare i manager.

L'architettura sfrutta `AppConfig`, che mantiene tutti i parametri di configurazione necessari per l'inizializzazione dei manager. L'uso di metodi statici nella factory riflette la natura utility-oriented del modulo, non è infatti necessario istanziare la factory stessa.

3.1.2.5 Modulo Routing

Il modulo Routing implementa il pattern **Front Controller**: centralizza la gestione di tutte le richieste HTTP dell'applicazione. Infatti questo modulo funge da punto di ingresso per tutte le richieste client, orchestrando il loro instradamento verso i controller appropriati. L'architettura del routing è organizzata secondo una struttura REST, dove le routes sono raggruppate logicamente in base al dominio di competenza (Auth, User, Operation, Notification, Radiograph).

Ogni route è configurata con il proprio metodo HTTP appropriato (GET, POST, PATCH) seguendo le convenzioni REST, e include la gestione dei parametri necessari, sia che provengano dal body della richiesta (`request.json`), dai parametri URL, o dai form data (`request.files`, `request.form`). L'architettura adottata permette una facile estensione del sistema.

3.1.2.6 Modulo Config

Il modulo Config è responsabile della gestione della configurazione del sistema attraverso la classe `AppConfig`. Questa classe segue il principio della configurazione esterna, consentendo di definire i parametri critici del sistema tramite variabili d'ambiente, pur garantendo la presenza di valori di default appropriati per una configurazione predefinita e robusta.

Anche in questo modulo, si è scelto di adottare una struttura organizzata in sezioni logiche, ciascuna dedicata a un ambito specifico. Le configurazioni includono:

- **Configurazioni d'ambiente**: per abilitare e controllare la modalità di debug.
- **Impostazioni CORS**: per gestire le richieste cross-origin in modo sicuro (6).
- **Percorsi delle credenziali**: necessari per integrare i servizi di Firebase e Google Cloud Storage.
- **Configurazioni SMTP**: per il servizio di invio email.
- **Percorsi dei modelli di machine learning**: per il caricamento e l'uso degli algoritmi di apprendimento automatico.

3.1.2.7 Modulo UI

Il modulo UI rappresenta il livello di presentazione del sistema ed è implementato seguendo il pattern architetturale **MVVM (Model-View-ViewModel)** attraverso il framework `Vue.js` (8). Questa architettura è una variante del pattern MVC (Model-View-Controller), infatti consente di separare gli aspetti dell'interfaccia grafica dalla logica del dominio, ma è progettato appositamente per semplificare la programmazione basata su eventi delle interfacce utente. In questo modo l'interfaccia non deve occuparsi della logica e grazie all'indipendenza delle varie componenti, è possibile rinnovare l'interfaccia senza modificarne il comportamento. Le principali componenti dell'architettura sono:

- **Model**, rappresenta il dominio dell'applicazione e il punto di accesso ai dati, quindi gestisce la logica dell'applicazione e di elaborazione dei dati. Comprende quindi il livello dei dati rappresentato dai repositories e dai data sources.
- **View**, racchiude gli UI Elements, rappresenta quindi la User Interface con cui l'utente interagisce, si occupa di visualizzare i dati e di ricevere le interazioni dell'utente.
- **ViewModel**, è rappresentato dagli State holders, fungono da centro di comunicazione tra il Model e la View, infatti interagiscono con il Model richiedendo i dati necessari alla View.

L'interfaccia utente è strutturata come una **Single Page Application (SPA)**, consentendo una navigazione fluida senza ricaricamento delle pagine e il caricamento asincrono delle risorse. Inoltre, il sistema gestisce lo stato dell'applicazione lato client, comunicando in modo efficiente con il backend attraverso chiamate API.

L'architettura è basata su componenti Vue, dove ogni componente rappresenta un'unità funzionale indipendente e riutilizzabile. La reattività dei dati è garantita da meccanismi come le proprietà reattive interne `data` e quelle passate dai componenti padre

ai figli `props`. La comunicazione tra i componenti avviene tramite eventi personalizzati per le interazioni figlio-padre e mediante un event bus per quelle tra componenti non collegati direttamente. Inoltre, il binding bidirezionale dei dati è implementato usando la direttiva "v-model".

L'interfaccia segue un design minimalista, orientato alla chiarezza e all'usabilità. Il layout è responsive grazie all'uso di un sistema di griglie flessibile, mentre la palette cromatica professionale basata su tonalità di blu trasmette affidabilità. La consistenza visiva è mantenuta attraverso l'uso di componenti riutilizzabili.

Per garantire prestazioni ottimali, il modulo UI utilizza il lazy loading dei componenti, il caching delle risorse statiche e l'ottimizzazione delle renderizzazioni tramite il Virtual DOM di Vue. La gestione delle dipendenze tra i componenti è stata pianificata per ridurre il carico computazionale e migliorare la reattività dell'applicazione.

Un sistema robusto di validazione e gestione degli errori è stato integrato nell'interfaccia. I form sono validati in tempo reale con feedback immediato per l'utente, mentre gli errori sono gestiti centralmente e comunicati con messaggi chiari. Per le operazioni asincrone, vengono utilizzati stati di loading accompagnati da feedback visivi che rendono evidente lo stato delle operazioni all'utente.

In conclusione, il modulo UI è stato progettato e sviluppato per garantire una user experience di alto livello, con un'architettura scalabile e un design moderno, rispettando le best practices consolidate.

3.1.2.8 Modulo Service

Il modulo `Service` del Server è fondamentale, in quanto definisce l'API del Server stesso e gestisce l'esposizione del servizio ai client. L'intero funzionamento dell'applicazione si basa sul meccanismo di *Remote Procedure Call (RPC)*: i client inviano richieste al Server e attendono una risposta. Ogni richiesta è caratterizzata da tre elementi fondamentali:

- **URL**, che identifica la risorsa o operazione richiesta.
- **Metodo**, che specifica l'operazione da eseguire.
- **Risposta**, che include uno status code standard e, in caso di successo, il risultato dell'operazione.

Come specificato anche prima, viene usato il protocollo HTTP per le comunicazioni e consente di eseguire operazioni *CRUD (Create, Read, Update, Delete)* attraverso i seguenti metodi:

- **POST**: per creare una nuova risorsa.
- **GET**: per leggere o recuperare dati.

- **PUT**: per aggiornare o sostituire una risorsa esistente.
- **PATCH**: per modificare parzialmente una risorsa.
- **DELETE**: per rimuovere una risorsa.

Il modulo `Service` si basa su librerie come `Axios` e il pacchetto `Firebase`. Ogni funzione all'interno del modulo rappresenta un'interfaccia per un endpoint specifico del server, fornendo una forma semplificata per inviare richieste e gestire le risposte.

Ogni funzione costruisce richieste HTTP che puntano agli endpoint definiti nel backend.

3.1.3 Struttura modulo Addestramento periodico

La rete neurale sfrutta un'infrastruttura cloud per gestire i processi di addestramento e preprocessamento dei dati. La struttura si basa su tre componenti principali:

- **Vertex AI**: viene utilizzato come piattaforma principale per l'esecuzione del notebook, consentendo una **gestione efficiente delle risorse di calcolo**. Questo approccio ha facilitato la prova delle pipeline, la verifica del corretto funzionamento e l'ottimizzazione dei vari step per rendere l'intero processo pronto per l'esecuzione periodica schedulata su Dataproc. Vertex AI, infatti permette nativamente una integrazione con gli altri servizi di Google Cloud, come Dataproc e Cloud Storage.
- **Bucket GCS**: Il bucket su Google Cloud Storage funge da archivio centrale per tutti i dati utilizzati nella rete. Contiene il **dataset originale**, che viene poi suddiviso in tre sottocartelle principali (train, val e test), così come i modelli salvati e i pesi generati durante l'addestramento. Inoltre, sempre nell'ottica della periodicità dell'addestramento, nel bucket vengono aggiute le nuove radiografie valutate positivamente dai medici, e pronte per addestrare la successiva versione del modello. Questo sistema di storage permette un accesso rapido ai dati e assicura che tutte le risorse siano centralizzate e disponibili per l'elaborazione su diversi servizi. Inoltre, GCS facilita il trasferimento e la condivisione dei dati tra diverse istanze e job di calcolo, garantendo al contempo la persistenza dei dati anche tra esecuzioni.
- **Dataproc**: Per l'automatizzazione dell'intero processo, è stato adottato Dataproc, che, tramite l'interfaccia di Vertex, permette di eseguire il notebook in modo periodico. Dataproc utilizza cluster configurabili per l'elaborazione distribuita dei dati e per l'addestramento del modello, offrendo un'efficienza ottimale grazie alla possibilità di scalare dinamicamente le risorse. Il progetto utilizza più worker per la parte di pre-processing, mentre utilizza una strategia *OneDeviceStrategy*, limitata al piano gratuito di Google Platform, che permette l'esecuzione dell'addestramento in un singolo worker. La struttura di Dataproc ha permesso di combinare flessibilità, potenza di calcolo e automazione, garantendo un flusso di lavoro efficiente e facile da scalare per addestramenti futuri.

- **WebApplication**: Recupera i pesi salvati su GoogleCloudStorage, al fine di aggiornare il modello disponibile per le predizioni dell'utente.

Di seguito, nella figura 12, è riportato il collegamento della rete con la struttura del sistema descritto nei punti precedenti.

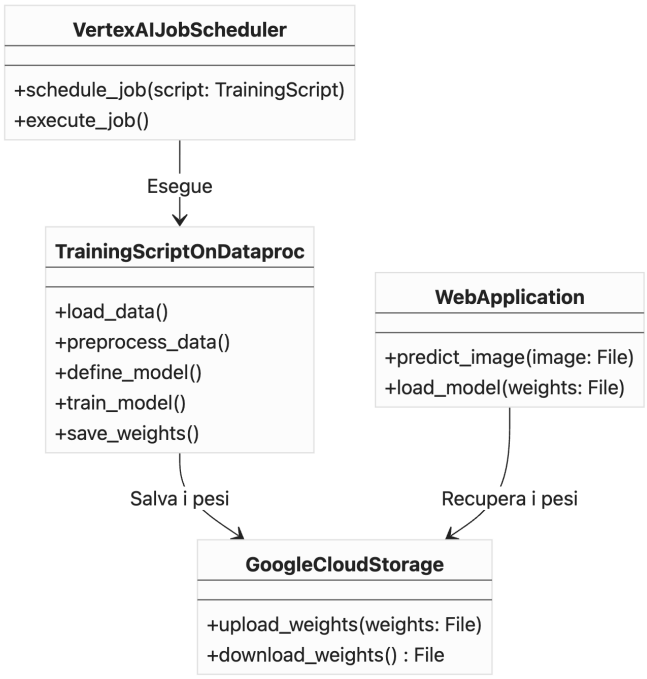

Figura 12: Diagramma delle classi: rete neurale

3.2 Comportamento

3.2.1 Comportamento modulo Controller

Il diagramma in Figura 13 illustra il flusso operativo generale dei controller all'interno del sistema. I controller sono responsabili della gestione delle richieste HTTP provenienti dal frontend, garantendo che ogni richiesta sia elaborata in modo corretto e che venga restituita una risposta appropriata.

Il processo inizia con la **ricezione della richiesta**, **seguita dalla sua validazione**. In questa fase, il sistema verifica che i dati siano conformi ai requisiti dell'endpoint richiesto; in caso contrario, restituisce un errore al client. Se la richiesta è valida, si passa alla **fase di elaborazione**.

Durante l'elaborazione, viene **verificata l'autorizzazione dell'utente** per accedere alla risorsa o eseguire l'operazione richiesta. Successivamente, il controller **esegue la**

logica di business, interagendo con i manager per gestire i dati e i servizi esterni. Una volta completata l'elaborazione, **i dati vengono formattati** per essere restituiti al client in un formato standard.

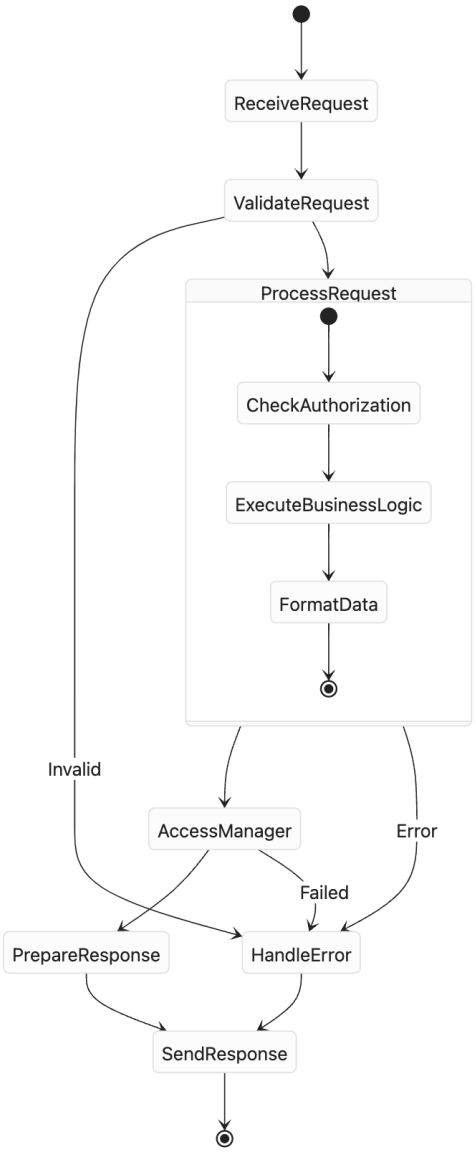

Figura 13: Diagramma delle attività: modulo Controller.

In caso di errori durante l'elaborazione, il sistema attiva un **flusso di gestione degli errori**, restituendo un messaggio di errore al client. Altrimenti, la risposta viene preparata e inviata, completando il flusso. Questo approccio garantisce modularità, separazione delle responsabilità e una gestione uniforme delle richieste e delle risposte,

facilitando la manutenzione e l'estensione del sistema.

3.3 Comportamento modulo Addestramento periodico

Il flusso di dati del notebook rappresenta un processo automatizzato e ciclico per l'addestramento di un modello di deep learning, orchestrato da **Vertex AI** e supportato da **Dataproc**. Con l'avvio delle computazioni logiche di **PySpark** viene avviato un **Job Scheduler**, che configura l'ambiente e avvia il cluster necessario per eseguire tutte le operazioni.

In primo luogo, il sistema verifica la presenza del dataset sul corrispondente Bucket su **Google Cloud Storage (GCS)**. Se la cartella dei dati non è presente, questi vengono scaricati, estratti da uno zip e caricati nel bucket in una struttura organizzata con sottocartelle per i set di train, validation e test. Una volta che i dati sono pronti, i percorsi delle immagini vengono raccolti e trasformati in un formato strutturato per permettere analisi e manipolazioni successive.

La **fase di preprocessing** inizia con l'estrazione delle etichette dai percorsi dei file e con la suddivisione del dataset in *training*, *validation* e *test*, assicurandosi che le distribuzioni siano coerenti. In questa fase, viene condotta un'analisi esplorativa che mostra la configurazione dei dati, inclusa la distribuzione delle immagini per classe o grado. Inoltre, il sistema bilancia le classi calcolando i pesi in base alla distribuzione dei dati, e applica tecniche di normalizzazione e data augmentation per ottimizzare l'efficacia dell'addestramento. Infine, i dati vengono salvati in formato TFRecord per ottimizzarne il caricamento e la gestione durante il training.

Il modello, basato su una rete **ResNet50 pre-addestrata**, viene configurato per supportare la classificazione in cinque classi. L'addestramento del modello avviene utilizzando una strategia **OneDeviceStrategy**, che sfrutta le risorse hardware disponibili, come i core multipli all'interno di un singolo dispositivo. Sebbene non si tratti di un approccio distribuito su più nodi, questa configurazione consente comunque di gestire il carico in modo efficace. Durante l'addestramento, i pesi del modello vengono periodicamente salvati sia localmente che su GCS, consentendo una conservazione sicura e pronta per futuri utilizzi.

Una volta completato l'addestramento, **il modello finale viene salvato** e caricato su GCS, insieme ai pesi aggiornati. Il sistema esegue una fase di valutazione per misurare le prestazioni sul set di test e calcola l'accuratezza complessiva del modello.
Il flusso termina con la chiusura automatizzata del **Job Scheduler**, che arresta il cluster e conclude il processo. Questo approccio consente un addestramento periodico e scalabile, che però non è stato possibile eseguire per mancanza di risorse con il piano gratuito disponibile.

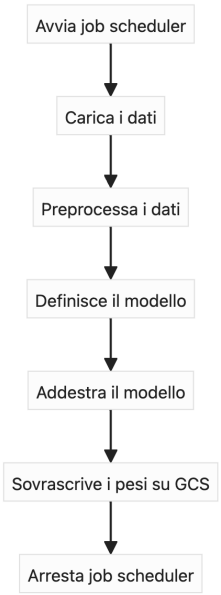

Figura 14: Diagramma delle attività: rete neurale.

3.4 Interazione

3.4.1 Modulo Core

Come mostrato in figura 15, il flusso operativo inizia con la creazione dell'app, che ha il compito di orchestrare l'intero processo di configurazione e inizializzazione del sistema.

Il primo passo prevede il **caricamento delle variabili di configurazione** del modulo Config, permettendo di leggere le variabili d'ambiente definite nel sistema e assicurando che tutti i parametri necessari siano correttamente caricati e disponibili per i moduli successivi.

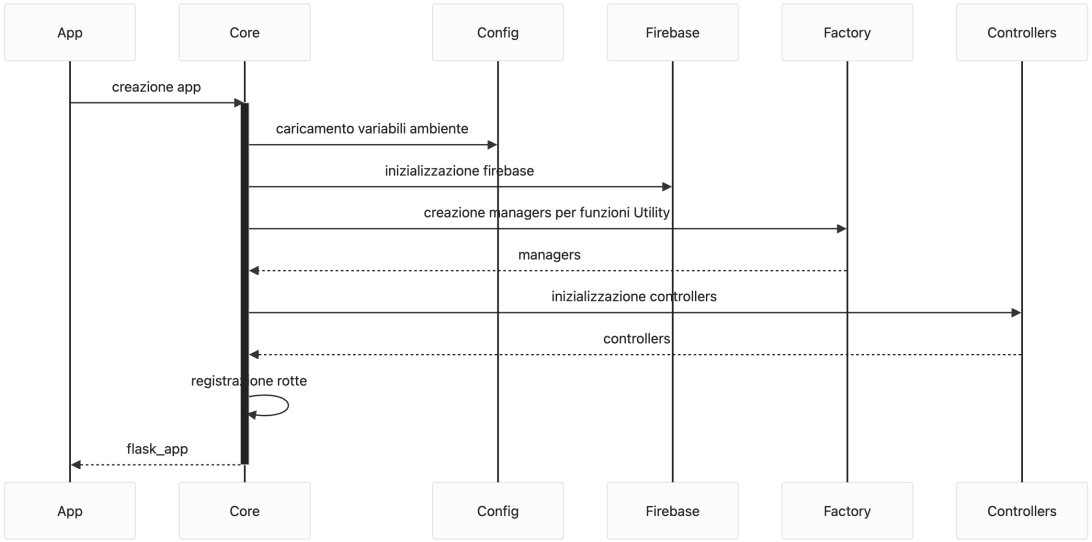

Figura 15: Diagramma di sequenza: modulo Core.

Dopodiché, il modulo Core si occupa dell'**inizializzazione dei servizi Firebase**, configurando il client utilizzando le credenziali specificate nel sistema. Questo passaggio garantisce una connessione sicura ai servizi di Firebase come Firestore e Firebase Authentication.

Dopo aver configurato i servizi esterni, il Core delega al modulo Factory la **creazione dei manager**. I manager, come il **FirestoreManager** e il **GCSManager**, rappresentano componenti chiave del sistema, e sono utilizzati per gestire operazioni specifiche sui dati e sui servizi esterni.

Dopo la creazione e configurazione dei manager, il Core si occupa dell'**inizializzazione dei controller**, ciascuno progettato per gestire un dominio specifico dell'applicazione. I controller vengono istanziati e collegati ai manager, che forniscono loro le funzionalità necessarie. Questa fase garantisce che ogni controller disponga delle risorse adeguate per svolgere il proprio ruolo all'interno della logica del sistema.

Infine, il Core **registra le rotte HTTP** attraverso il modulo Routing, collegando ciascun controller agli endpoint definiti. Questo processo centralizza la gestione delle richieste, rendendo disponibili le API REST per gli utenti del sistema.

3.4.2 Modulo Factory

Il modulo Factory centralizza la creazione e la configurazione dei manager necessari per il sistema, garantendo una gestione ordinata delle dipendenze e delle risorse.

Il processo inizia con il recupero delle variabili di configurazione tramite l'importazione del modulo Config. Successivamente, vengono inizializzati e creati i seguenti manager:

- **firestore**: il manager Firestore viene inizializzato creando un'istanza del client Firestore tramite la libreria `firebase_admin`. Questo manager è responsabile delle operazioni CRUD sul database Firestore.
- **gcs**: il manager Google Cloud Storage (`GCSManager`) è inizializzato specificando il nome del bucket GCS. È responsabile del caricamento e della gestione dei file non strutturati, come le radiografie e i pesi del modello.
- **model**: il modello di machine learning viene caricato sfruttando il `GCSManager` e il percorso specificato nel modulo Config. Una volta caricato, viene utilizzato per creare un'istanza del `ModelManager`, che gestisce tutte le operazioni legate al modello, inclusa l'elaborazione delle immagini e la predizione.
- **email**: l'`EmailManager` è inizializzato configurando le credenziali SMTP per l'invio delle email. Questo manager utilizza il servizio SMTP per inviare notifiche e comunicazioni agli utenti del sistema in modo affidabile.

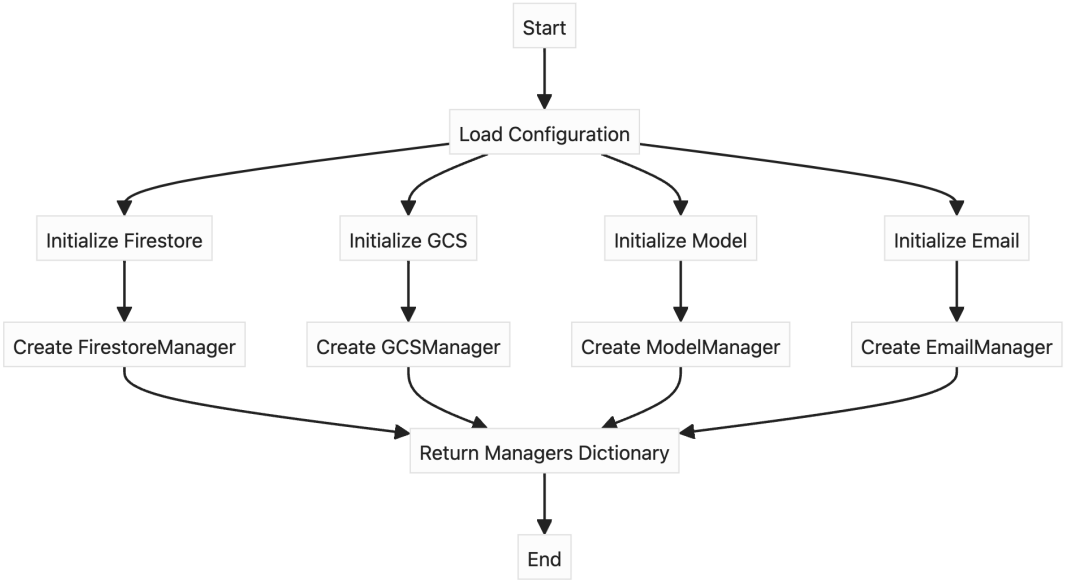

Figura 16: Diagramma di flusso: modulo factory

3.4.3 Modulo Routing

Il modulo Routing rappresenta il punto di accesso per tutte le richieste HTTP dell'applicazione. Questo modulo centralizza la gestione delle API REST, migliorando l'organizzazione e la coerenza del sistema.

La sua funzione principale è quella di instradare ogni richiesta al controller corretto, in modo che ogni operazione venga gestita dalla logica di business appropriata. Le rotte sono state organizzate in base al dominio di competenza:

- **auth**: comprende tutte le operazioni legate all'autenticazione e all'autorizzazione degli utenti, come login, registrazione e verifica dell'email.
- **user**: comprende tutte le operazioni relative alla gestione degli utenti, come il recupero e l'aggiornamento delle informazioni personali.
- **notification**: comprende tutte le operazioni per la gestione delle notifiche.
- **operation**: comprende tutte le operazioni per la gestione della pianificazione di interventi chirurgici.
- **radiograph**: comprende tutte le operazioni per la gestione delle radiografie, come il caricamento, l'elaborazione e la predizione.

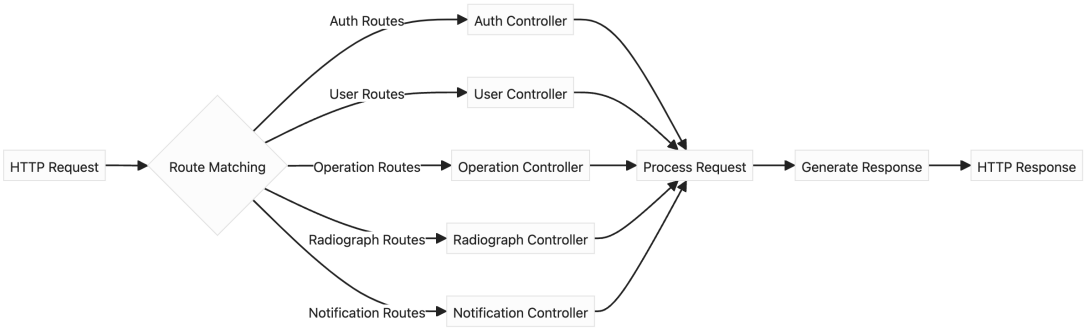

Figura 17: Diagramma di flusso: modulo Routing

Dopo che il controller appropriato ha processato la richiesta, il modulo Routing si occupa di generare una risposta strutturata, che viene restituita al client tramite il protocollo HTTP.

3.4.4 Modulo UI

Il modulo UI, illustrato in Figura 18, segue l'**architettura MVVM**, garantendo una chiara separazione tra logica e presentazione. Ogni interazione dell'utente, come un clic, è catturata dalla **View** e gestita dal **ViewModel**, che aggiorna i dati locali e, se necessario, invia richieste al backend tramite il **Service Layer**.

Le risposte del backend aggiornano il **Model**, e il **ViewModel** aggiorna la **UI**, mostrando i cambiamenti. Questo flusso assicura un'interfaccia reattiva e una gestione efficace delle operazioni asincrone, migliorando la manutenibilità e l'esperienza utente.

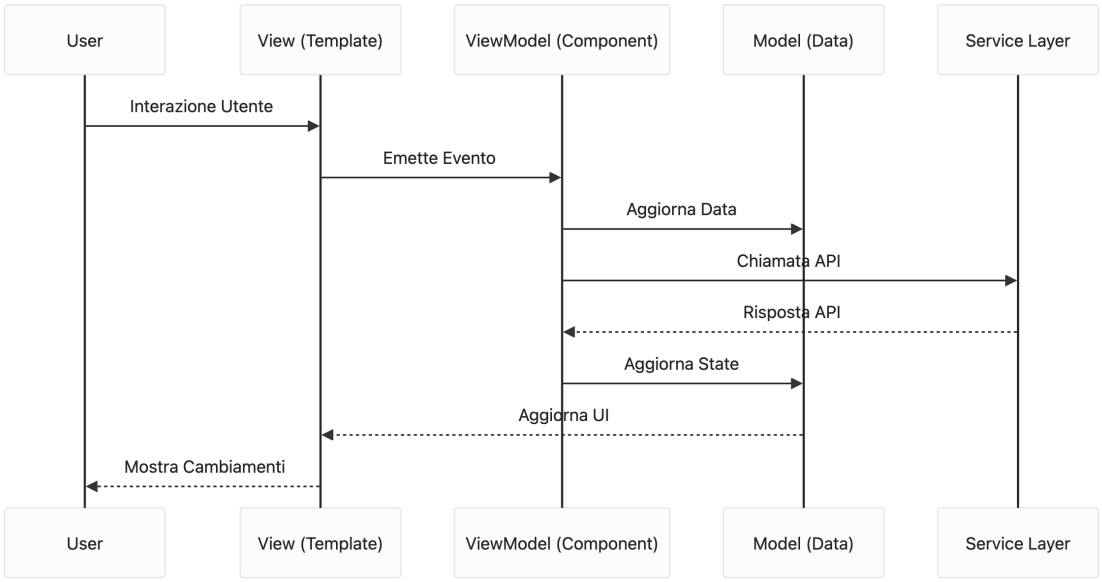

Figura 18: Diagramma di sequenza: modulo UI.

3.4.5 Modulo Addestramento periodico

Come mostrato nella figura 19, l'**automazione** del processo parte con l'avvio schedulato del notebook tramite Vertex AI su Dataproc. Durante l'esecuzione, lo script utilizza i dati e le risorse di calcolo disponibili per addestrare il modello.

Al termine dell'addestramento, **i pesi del modello vengono salvati** nel bucket di Google Cloud Storage. Successivamente, se il nuovo modello dimostra prestazioni superiori rispetto a quello attualmente in uso sul sito, quest'ultimo viene sostituito con la versione aggiornata. Da quel momento, tramite l'interfaccia web, sarà possibile effettuare predizioni con una maggiore precisione e affidabilità.

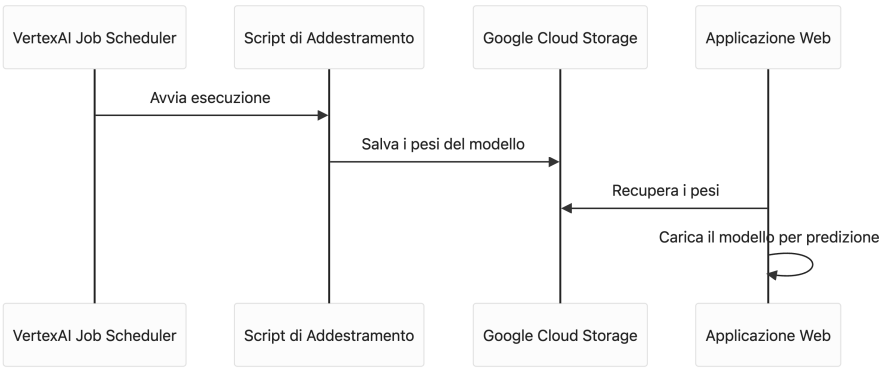

Figura 19: Diagramma di sequenza: rete neurale

4 Dettagli implementativi

4.1 Tecnologie

Sono di seguito riportate le principali tecnologie utilizzate per svolgere l'elaborato.

Node.js è stato utilizzato come runtime JavaScript lato server per lo sviluppo del frontend (7). Grazie alla sua architettura asincrona e basata su eventi, ha permesso di gestire in modo efficiente le richieste degli utenti e la comunicazione con il backend tramite API REST.

Flask è stato utilizzato come framework per la gestione delle API lato backend. Ha permesso di definire una serie di endpoint REST per gestire le varie richieste del client. Le route sono state configurate per rispondere a richieste HTTP specifiche, integrando i controller che implementano la logica applicativa.

Vue.js un framework JavaScript noto per la sua facilità di integrazione e leggerezza, è stato sfruttato per la creazione dell'interfaccia utente. Ogni elemento dell'interfaccia è stato suddiviso in componenti autonomi e modulari, come il caricamento delle radiografie o la visualizzazione delle diagnosi. Inoltre, la presenza di un `Vue Router` integrato ha facilitato la navigazione tra le diverse pagine del sistema, migliorando l'organizzazione e la fluidità dell'esperienza utente.
La scelta di Vue.js rispetto ad altre tecnologie come Angular o React è stata dettata dalla sua sintassi intuitiva, dalla flessibilità nella gestione dei componenti e dalla crescente popolarità tra gli sviluppatori, che lo rende particolarmente adatto a progetti moderni e scalabili.

Axios è una libreria di JavaScript utilizzata per semplificare la gestione delle comunicazioni HTTP tra il frontend e il backend. Grazie alla sua sintassi chiara ed intuitiva, ha permesso di effettuare richieste asincrone in modo efficiente (1).

Javascript per gestire l'interattività dell'interfaccia utente e le comunicazioni asincrone con il backend tramite chiamate API. Grazie alla sua versatilità, JavaScript si integra perfettamente con librerie e framework come `Vue.js` e `Axios`.

Firebase Authentication è stato utilizzato per gestire la registrazione, il login e l'autenticazione degli utenti. Questo servizio offre un sistema di autenticazione sicuro basato su token, garantendo che solo utenti autorizzati possano accedere al sistema.
Le principali funzionalità di Firebase Authentication implementate nel progetto includono:

- **Autenticazione tramite email e password**: gli utenti possono registrarsi con credenziali univoche e accedere in modo sicuro.

- **Verifica dell'email**: durante la registrazione, viene inviato un link di conferma per verificare l'indirizzo email dell'utente.
- **Gestione dei tentativi di login falliti e reset della password**: se un utente supera il limite massimo di tentativi di accesso (6), l'account viene temporaneamente bloccato e viene inviata un'email automatica con un link per il reset della password.

Firestore è un database NoSQL utilizzato per memorizzare dati strutturati legati agli utenti, alle operazioni pianificate e alle notifiche. Firestore è stato scelto per la sua semplicità di configurazione, l'integrazione nativa con **Firebase Authentication** e la capacità di garantire un'elevata disponibilità e affidabilità anche in applicazioni distribuite.

Google Cloud Storage è un servizio di storage scalabile e altamente affidabile offerto da Google Cloud Platform. Viene utilizzato per l'archiviazione di file e dati non strutturati come radiografie, heat map e file di testo con i risultati delle predizioni. Inoltre, sono salvati anche architettura e pesi del modello di `ResNet50`.

Docker per garantire un'implementazione modulare e facilmente gestibile. Il sistema è stato containerizzato e distribuito su una macchina virtuale ospitata su **Google Compute Engine** (4).

Per l'archiviazione di file e dati non strutturati, il sistema utilizza **Google Cloud Storage**, un servizio di storage scalabile e altamente affidabile offerto da Google Cloud Platform. I tipi di file salvati nel bucket includono immagini, come radiografie e heat map, file di testo con i risultati delle predizioni. Inoltre, sono salvati anche i pesi del modello di `ResNet50`

4.2 Dettagli implementativi Containerizzazione e Deployment

Per garantire un'implementazione modulare e facilmente gestibile, il sistema è stato containerizzato utilizzando **Docker** e distribuito su una macchina virtuale ospitata su **Google Compute Engine**. Questa configurazione permette di separare i diversi componenti del sistema in `container`, semplificando il deployment e la gestione.

4.2.1 `Dockerfile` per il frontend

Il Dockerfile per il frontend è basato su Node.js. Dopo aver impostato la directory di lavoro, i file `package.json` e `package-lock.json` vengono copiati per installare le dipendenze tramite `npm install`. Infine viene costruita l'applicazione Vue.js tramite `npm run build` esponendo la porta 8080 per permettere l'accesso all'applicazione.

Listing 1: Dockerfile per il frontend

```
# Usa un'immagine base di Node.js
FROM node:14

# Imposta la directory di lavoro
WORKDIR /osteoarthritis-project/frontend

# Copia il file package.json e package-lock.json
COPY package*.json ./

# Installa le dipendenze
RUN npm install

# Copia il resto del codice sorgente
COPY . .

# Costruisce il progetto Vue.js
RUN npm run build

# Espone la porta che l'applicazione utilizzer
EXPOSE 8080

# Comando per avviare il server
CMD ["npm", "run", "serve"]
```

4.2.2 `Dockerfile` per il backend

Il Dockerfile per il backend è basato su un'immagine di Python 3.9. Dopo l'installazione dei pacchetti necessari per OpenCV, viene impostata la directory di lavoro e copiati i file del progetto. Successivamente, viene eseguita l'installazione delle dipendenze Python tramite il file `requirements-vm.txt`. Il Dockerfile espone la porta 5000, necessaria per l'esecuzione del server Flask, e avvia l'applicazione con il comando `python app.py`.

Listing 2: Dockerfile per il backend

```
FROM python:3.9

# Aggiunge i pacchetti necessari per OpenCV
RUN apt-get update && apt-get install -y \
    libgl1-mesa-glx \
    libglib2.0-0 \
    && rm -rf /var/lib/apt/lists/*

# Imposta la directory di lavoro
WORKDIR /osteoarthritis-project/backend/app

# Copia i file del progetto nella directory di lavoro
COPY . .

# Installa le dipendenze dal file requirements.txt
RUN pip install --no-cache-dir -r requirements-vm.txt
```

```
# Espone la porta 5000 per il server Flask
EXPOSE 5000

# Comando per avviare l'applicazione
CMD ["python", "./app.py"]
```

4.2.3 Orchestrazione dei container: `docker-compose.yml`

Il file `docker-compose.yml` definisce due container separati, uno per il backend e uno per il frontend del progetto. Il container del backend espone la porta 5000, mentre quello del frontend espone la porta 8080.

Listing 3: docker-compose.yml

```
version: "3"
services:
  backend:
    image: andyalemonta/backend-docker-image:latest
    container_name: backend_container
    restart: always
    ports:
      - "5000:5000"
    networks:
      - app-network

  frontend:
    image: andyalemonta/frontend-docker-image:latest
    container_name: frontend_container
    restart: always
    ports:
      - "8080:8080"
    networks:
      - app-network

networks:
  app-network:
    driver: bridge
```

4.3 Dettagli implementativi Backend

4.3.1 Inizializzazione App

Come spiegato anche in precedenza, il modulo Core gestisce l'inizializzazione dell'app Flask, configurando l'ambiente e le dipendenze. La funzione `create_app` crea l'istanza di Flask e configura CORS per le richieste da origini specifiche. Utilizzando il pattern Factory, i managers vengono creati tramite `ManagerFactory.create_managers` e iniettati nei controller, come `AuthController`, `UserController` e altri, per gestire la logica di business. Infine, le route API sono registrate tramite `register_routes`, associando

ogni endpoint alla funzione del controller corrispondente. Questo approccio garantisce modularità, scalabilità e facilità di configurazione.

Listing 4: Operazioni di inizializzazione dell'app

```
def create_app():
    app = Flask(__name__)

    # Set Google Cloud credentials environment variable
    os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = AppConfig.
        GCS_CRED_PATH

    # Configure CORS using AppConfig
    CORS(app, resources={r"/*": {"origins": AppConfig.CORS_ORIGIN}})

    # Inizializza managers
    managers = ManagerFactory.create_managers(app.config)

    # Inizializza controllers
    controllers = {
        'auth': AuthController(managers),
        'user': UserController(managers),
        'operation': OperationController(managers),
        'notification': NotificationController(managers),
        'radiograph': RadiographController(managers)
    }

    # Registra routes
    register_routes(app, controllers)

    return app
```

4.3.2 Connessione a Google Cloud Storage

All'interno del modulo Utility, nel manager dedicato a Google Cloud Storage, viene configurata la connessione al servizio. La classe `GCSManager` si occupa dell'inizializzazione del client di Google Cloud Storage, utilizzando le credenziali specificate nel percorso configurato in `AppConfig.GCS_CRED_PATH`. In caso di errore durante l'inizializzazione, viene sollevata un'eccezione personalizzata `GCSManagerException`.

Listing 5: Connessione a Google Cloud Storage

```
@dataclass
class BlobInfo:
    """Classe di supporto per informazioni sui blob."""
    name: str
    url: Optional[str]
    created_at: datetime
    content_type: Optional[str]

class GCSManagerException(Exception):
    """Classe personalizzata per le eccezioni del GCS Manager."""
```

```
    pass

class GCSManager:
    """
    Gestore per le operazioni su Google Cloud Storage.
    Fornisce metodi per operazioni CRUD e funzionalit  specifiche per il
        dominio.
    """

    def __init__(self, bucket_name: str):
        try:
            credentials = service_account.Credentials.
                from_service_account_file(
                 AppConfig.GCS_CRED_PATH
            )
            self.storage_client = storage.Client()
            self.bucket = self.storage_client.bucket(bucket_name)
            self.bucket_name = bucket_name
        except Exception as e:
            raise GCSManagerException(f"Errore di inizializzazione: {str(
                e)}")
```

4.3.3 Caricamento del modello pre-addestrato

La funzione `load_model` del `GCSManager` consente di caricare un modello pre-addestrato direttamente dal bucket di Google Cloud Storage. Il codice scarica i dati del modello in memoria utilizzando il metodo `download_to_file` e li carica direttamente dal buffer tramite la libreria `h5py`. In questo modo, il modello viene caricato senza la necessità di salvarlo su disco, ottimizzando l'efficienza nella gestione delle risorse. In caso di errore durante il processo, viene sollevata un'eccezione personalizzata `GCSManagerException`.

Listing 6: Caricamento del modello pre-addestrato

```
    def load_model(self, model_path):
        try:
            # Scarica i dati in memoria
            blob = self.bucket.blob(model_path)
            model_bytes = io.BytesIO()
            blob.download_to_file(model_bytes)
            model_bytes.seek(0)

            # Carica il modello direttamente dal buffer
            with h5py.File(model_bytes, 'r') as h5file:
                model = load_model(h5file)

            print("Modello caricato correttamente dalla memoria!")
            return model

        except Exception as e:
            raise GCSManagerException(f"Errore nel caricamento del
                modello: {str(e)}")
```

4.3.4 Esempio di rotta: `get_patient_radiographs()`

Il codice sotto-riportato mostra un esempio di rotta **Flask** utilizzata per il recupero delle radiografie associate a un determinato paziente. Questa rotta viene esposta tramite un endpoint RESTful e si basa sul controller `RadiographController` per gestirne la logica.

Listing 7: Rotta di recupero delle radiografie

```
def register_routes(app, controllers):
    # Auth Routes
    ...
    # User Routes
    ...
    # Operation Routes
    ...
    # Notification Routes
    ...
    # Radiograph Routes
    @app.route('/api/patients/<patient_id>/radiographs', methods=['GET'])
    def get_patient_radiographs(patient_id):
        return controllers['radiograph'].get_patient_radiographs(
            patient_id)

    ...

    return app
```

All'interno della funzione `get_patient_radiographs()`, viene sfruttata una funzionalità del **GCSManager**, rappresentata dal metodo `list_patient_radiographs()` per ottenere l'elenco completo delle radiografie associate a un paziente.

Dopodiché, i dati recuperati vengono restituiti al client sotto forma di una risposta JSON.

Listing 8: Funzione get_patient_radiographs() contenuta in RadiographController

```
class RadiographController:
    def __init__(self, managers):
        self.gcs_manager = managers['gcs']
        self.model_manager = managers['model']
        self.firestore_manager = managers['firestore']

    def get_patient_radiographs(self, patient_id):
        try:
            radiographs = self.gcs_manager.list_patient_radiographs(
                patient_id)
            response = [{
                "url": rad.url,
                "name": rad.name,
                "date": rad.created_at.strftime("%Y-%m-%d")
            } for rad in radiographs]
            return jsonify(response), 200
        except Exception as e:
```

```
        return jsonify({"error": str(e)}), 500
```

4.3.5 Gestione del Cross Origin Resource Sharing

Il progetto utilizza una configurazione CORS per consentire al frontend e al backend di comunicare in sicurezza, rispettando le politiche di sicurezza del browser.

Durante lo sviluppo in locale, il backend era configurato per accettare richieste provenienti da `http://localhost:8080`, come mostrato nel seguente listato:

Listing 9: Gestione del CORS localmente

```
from flask_cors import CORS

CORS(app, resources={r"/*": {"origins": "http://localhost:8080"}})
```

Successivamente, in fase di deployment, il valore del dominio è stato aggiornato per accettare richieste dalla macchina virtuale ospitata su Google Compute Engine, con l'indirizzo IP pubblico `34.56.199.129`:

Listing 10: Gestione del CORS sulla vm

```
from flask_cors import CORS

CORS(app, resources={r"/*": {"origins": "http://34.56.199.129:8080"}})
```

4.4 Dettagli implementativi Frontend

4.4.1 Gestione delle chiamate API: `api-service.js`

Il file `api-service.js` centralizza le chiamate API per l'applicazione, usando Axios per effettuare richieste HTTP al backend. Contiene funzioni per operazioni comuni come il login, il caricamento delle radiografie e il recupero dei pazienti associati a un dottore. Questo approccio facilita la gestione delle comunicazioni con il server, mantenendo il codice modulare e facilmente estendibile.

Listing 11: api-service.js

```
import axios from "axios";

const API_URL = "http://127.0.0.1:5000"; // URL backend locale
//const API_URL = "http://34.56.199.129:5000"; // URL pubblico VM Compute
    Engine

// Funzione per ottenere le radiografie di un paziente
export const getRadiographs = async (patientId) => {
  try {
    const response = await axios.get(
      `${API_URL}/api/patients/${patientId}/radiographs`
    );
```

```
    return response.data;
  } catch (error) {
    console.error("Errore nel recupero delle radiografie:", error);
    throw error;
  }
};
};
```

4.5 Dettagli dell'addestramento automatico

4.5.1 Struttura del Dataset

Nella parte riguardante il modello da addestrare, i dati utilizzati sono costituiti da radiografie archiviate su Google Cloud Storage (GCS), suddivise per grado di osteoartrite secondo la scala Kellgren-Lawrence. Nello specifico, il dataset iniziale è archiviato in un file .zip all'interno di un bucket su GCS.

Il codice come prima cosa verifica la presenza di una struttura organizzata in cartelle per train, validation e test. Se questa struttura non è presente, il programma richiede che sul bucket sia presente il file .zip del Dataset sopra citato. Dopodichè lo scarica, lo decomprime e le immagini vengono caricate nuovamente sul bucket, organizzate in modo appropriato. Questa procedura iniziale garantisce un dataset ben organizzato.

Successivamente, come mostrato nel listato 12, viene aperta una sessione Spark, che consente di gestire i percorsi delle immagini, caricandoli in un DataFrame Spark per un'elaborazione parallela.

Listing 12: Creazione della Sessione Spark

```
# Crea una sessione Spark
spark = SparkSession.builder \
    .appName("KneeOsteoarthritisPreprocessing") \
    .config("spark.yarn.appMasterEnv.PYSPARK_PYTHON", "/opt/conda/bin/
        python3.10") \
    .config("spark.executorEnv.PYSPARK_PYTHON", "/opt/conda/bin/python3
        .10") \
    .getOrCreate()


# Recupera i percorsi delle immagini dalla cartella /Dataset su GCS
blobs = bucket.list_blobs(prefix="Dataset/")
image_paths = [blob.name for blob in blobs if blob.name.endswith('.png')]

print(f"Numero totale di immagini trovate: {len(image_paths)}")
```

Nell'utilizzo finale dell'applicativo, quando un chirurgo conferma la correttezza di una predizione, la corrispondente radiografia viene aggiunta al bucket. Questo approccio

consente di arricchire progressivamente l'insieme delle radiografie, migliorando il potenziale del sistema.

In seguito, il dataset è suddiviso in training, validation e test set. La suddivisione segue proporzioni del 67.5% per il training, 22.5% per la validation e 10% per il test, seguendo valori determinati grazie a test precedenti, che si sono dimostrati efficaci per garantire un buon bilanciamento e una valutazione affidabile del modello.
Va però sottolineato che l'obiettivo principale di questo rete non sia ottimizzare le prestazioni del modello, ma dimostrare la possibilità di utilizzare Google Cloud Platform per task di questo tipo, sfruttando anche PySpark.

4.5.2 Sviluppo del modello

Una volta effettuate le diverse operazioni di pre-processing, per ottimizzare l'elaborazione nelle fasi successive, le immagini vengono convertite in formato TFRecord. Questo formato è preferibile per la parte di addestramento, rispetto al formato byte dei DataFrame Spark, in quanto consente un caricamento più rapido e un accesso sequenziale ottimizzato. Dopo la conversione, i file TFRecord vengono caricati su GCS per garantire un accesso rapido durante la fase di training, migliorandone l'efficienza.

A questo punto si è passati alla definizione del modello vero e proprio. La scelta della rete neurale convoluzionale ResNet50, pre-addestrata su ImageNet, è stata dettata dalla sua capacità di estrarre feature gerarchiche dalle immagini e dall'opportunità di sfruttare il transfer learning per adattarla al compito specifico della classificazione secondo la scala Kellgren-Lawrence.

Per configurare il modello, i layer pre-addestrati della ResNet50 sono stati mantenuti come estrattori di feature e congelati inizialmente per preservare i pesi ottimizzati su ImageNet. A questi layer è stato aggiunto un blocco finale personalizzato, sfruttando la tecnica del `Fine Tuning`, composto da un layer di pooling globale e un layer denso con cinque neuroni, ciascuno corrispondente a una classe del problema. La funzione di attivazione softmax è stata scelta per generare probabilità normalizzate sulle cinque classi, garantendo così un output interpretabile per la classificazione.

Un elemento fondamentale del processo è stata la definizione della **Strategy OneDeviceStrategy**. Sebbene il preprocessing dei dati venga eseguito in modo distribuito per migliorare l'efficienza, si è deciso di non distribuire l'addestramento del modello. Test precedenti avevano infatti evidenziato che la distribuzione della fase di training poteva introdurre instabilità nel processo di ottimizzazione. La OneDeviceStrategy ha consentito di ottenere una maggiore stabilità durante l'addestramento, bilanciando l'efficienza computazionale e la robustezza del modello.

Listing 13: Definizione della Strategy

```
# Importazione delle librerie necessarie
import tensorflow as tf
from tensorflow.keras.optimizers import Adam

# Definizione della Strategy
strategy = tf.distribute.OneDeviceStrategy(device="/CPU:0")

# Definizione della shape dell'immagine e del numero di classi
img_shape = (224, 224, 3)
num_classes = 5

# Costruzione del modello all'interno del contesto della strategy
with strategy.scope():
    # Caricamento il modello ResNet50 pre-addestrato su ImageNet
    base_model = tf.keras.applications.ResNet50(
        input_shape=img_shape,
        include_top=False,
        weights="imagenet",
    )

    # Congelamento dei layer del modello base
    base_model.trainable = True

    # Costruzione del modello completo
    inputs = tf.keras.Input(shape=img_shape)
    x = base_model(inputs, training=True)
    x = tf.keras.layers.GlobalAveragePooling2D()(x)
    x = tf.keras.layers.Lambda(lambda inputs: tf.nn.dropout(inputs, rate
        =0.2))(x)
    outputs = tf.keras.layers.Dense(num_classes, activation="softmax")(x)

    # Definizione del modello
    model_ft = tf.keras.Model(inputs, outputs)

    # Compilazione del modello
    model_ft.compile(
        optimizer=Adam(learning_rate=0.0001),
        loss="categorical_crossentropy",
        metrics=["accuracy"],
    )

# Visualizzazione della struttura del modello
model_ft.summary()
```

Prima di iniziare l'addestramento, è stato affrontato il problema del bilanciamento delle classi. Poiché alcune di esse erano rappresentate da un minor numero di immagini, sono stati calcolati i pesi delle classi utilizzando una funzione della libreria Scikit-learn. Questi pesi sono stati integrati nella funzione di perdita per ridurre il rischio di bias verso le classi più frequenti. L'algoritmo di ottimizzazione scelto è stato Adam, configurato con un learning rate iniziale ridotto per garantire una convergenza più stabile durante il fine-tuning.

Un altro aspetto importante del processo è stata l'integrazione di callback per il salvataggio dei pesi del modello a ogni epoca. I checkpoint sono stati caricati automaticamente su Google Cloud Storage, consentendo di riprendere l'addestramento in caso di interruzioni o errori e garantendo una gestione robusta dell'intero processo.

Durante il training, il modello è stato valutato periodicamente sul validation set per monitorare l'overfitting e ottimizzare i parametri di addestramento. Al termine, è stato testato sul test set per calcolare metriche come accuratezza e perdita. I risultati riflettono un numero limitato di epoche, poiché l'obiettivo principale era dimostrare la fattibilità di Vertex AI nella gestione della rete neurale. L'attenzione si è quindi concentrata sull'integrazione efficace delle operazioni di pre-processing e di addestramento su Google Cloud Platform, piuttosto che sul miglioramento delle prestazioni del modello.

5 Autovalutazione/Validazione

5.1 Testing Manuale

Il sistema è stato **testato manualmente** dai tre membri del gruppo e dal primario di ortopedia con cui il gruppo collabora. Durante questa fase, sono state verificate manualmente le principali funzionalità dell'applicazione, controllando che i vari flussi (registrazione, login, caricamento radiografie, ecc...) risultassero utilizzabili e privi di errori.

5.2 Testing Automatizzato

Per eseguire test automatizzati sulle principali rotte e funzioni del sistema è stato utilizzato **Pytest**, un framework per il testing in Python. In particolare, è stato creato un file chiamato `conftest.py`, che viene interpretato automaticamente da Pytest come configurazione generale.

Listing 14: Contenuto del file `conftest.py`

```
import pytest
import sys
import os
from unittest import mock
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "
    ../backend")))

# Fixture per il client Flask
@pytest.fixture
def client():
    """Fixture per il client Flask."""
    from app import app
    #print("Registered routes:", app.url_map)
```

```
    app.config['TESTING'] = True
    with app.test_client() as client:
        yield client
```

Nel codice riportato nel listing si nota come il file di configurazione è stato utilizzato per:

- **configurare l'ambiente di test**: aggiungendo ai path di sistema la cartella contenente il backend.
- **fornire la fixture** `client`: che crea un'istanza dell'applicazione Flask in modalità test.

Grazie a questa configurazione i vari test possono inviare richieste HTTP senza dover replicare ogni volta la creazione e configurazione del client.

Per garantire una verifica completa del comportamento del sistema, i test automatizzati sono stati raggruppati in base alle principali aree funzionali dell'applicazione. Ogni gruppo di test si concentra su un preciso ambito (ad esempio la gestione dell'autenticazione, delle notifiche o delle operazioni), in modo da coprire tutti i possibili casi rilevanti.

Per rendere i test più compatti, estensibili ed efficaci, è stato utilizzato il decoratore `pytest.mark.parametrize` di **Pytest**. Questo approccio ha permesso di eseguire più test con diverse combinazioni di parametri senza dover duplicare il codice, migliorando così la leggibilità e la manutenzione.

Di seguito è riportato un test che verifica il comportamento dell'endpoint `/get-patient-operations`:

Listing 15: Esempio di test per il recupero delle operazioni

```
@pytest.mark.parametrize(
    "patient_id, mock_side_effect, mock_return_docs, expected_status_code
        , expected_value_substr",
    [
        # Dati di test
        ("pat123", None, [{"id": "op1", "description": "Operazione 1"}, {
            "id": "op2", "description": "Operazione 2"}], 200, "op1"),
        ("patError", Exception("Errore Firestore"), None, 500, "Errore
            Firestore"),
    ]
)

@patch("utils.firestore_utils.FirestoreManager.query_documents")
def test_get_patient_operations(
    mock_query_documents,
    patient_id,
    mock_side_effect,
    mock_return_docs,
    expected_status_code,
```

```
    expected_value_substr,
    client
):
    # Configurazione del mock
    if mock_side_effect:
        mock_query_documents.side_effect = mock_side_effect
    else:
        mock_query_documents.return_value = mock_return_docs or []

    # Costruzione URL e richiesta GET
    url = f"/api/patients/{patient_id}/operations"
    response = client.get(url)
    json_data = response.get_json()

    # Asserzioni
    assert response.status_code == expected_status_code, json_data
    if expected_status_code == 200:
        # Caso di successo -> JSON con lista di operazioni
        assert isinstance(json_data, list)
        assert any(expected_value_substr in str(op) for op in json_data)
    else:
        # Caso di errore -> JSON con chiave "error"
        assert "error" in json_data
        assert expected_value_substr in json_data["error"]
```

Viene riportato di seguito l'elenco di tutti i test effettuati:

Controller	Test
Auth	
	test_register_user
	test_login_parametrized
	test_check_email_verification
	test_verify_email
	test_reset_password
	test_send_reset_email
	test_decrement_attempts
	test_get_attempts_left
Notification	
	test_send_notification
	test_get_notifications
	test_mark_notification_as_read
Operation	
	test_add_operation
	test_get_patient_operations
Radiograph	
	test_get_patient_radiographs
	test_download_radiograph
	test_upload_to_dataset
	test_get_radiographs_info
	test_predict
User	
	test_get_user
	test_update_user
	test_get_doctors
	test_get_patients
	test_get_patients_from_doctor

Tabella 1: Elenco dei test divisi per tipologia di controller

6 Istruzioni per il deployment

Il sistema è stato distribuito utilizzando **Docker** su una macchina virtuale configurata su **Google Compute Engine**.

6.1 Creazione delle immagini Docker

Le immagini Docker per il frontend e il backend sono state create direttamente sulla VM, utilizzando i Dockerfile descritti in precedenza, e contengono tutti gli elementi necessari per eseguire i rispettivi servizi, inclusi i file sorgenti, le dipendenze e le configurazioni. Sono di seguito riportati gli output dei due comandi utilizzati per costruire le immagini:

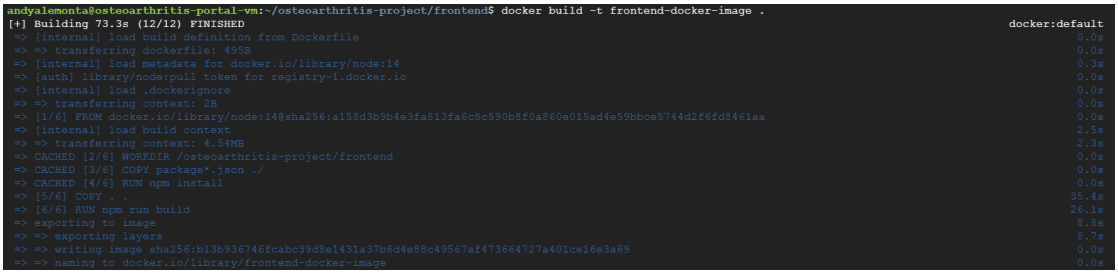

Figura 20: Costruzione dell'immagine Docker per il frontend

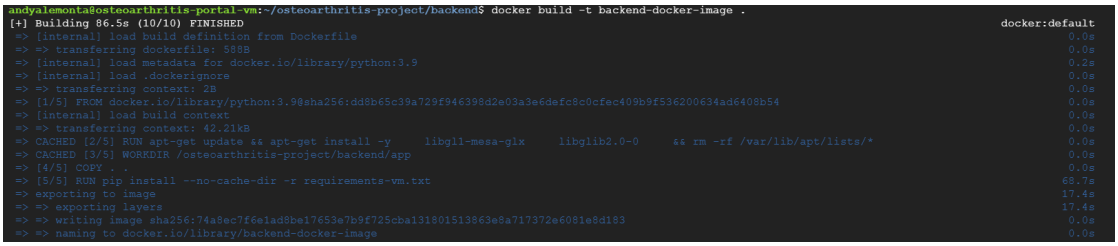

Figura 21: Costruzione dell'immagine Docker per il backend

6.2 Tagging delle immagini Docker

Dopo aver creato le immagini, è stato eseguito il comando di **tagging** per assegnare un'etichetta univoca alle immagini:

```
docker tag frontend-docker-image:latest andyalemonta/frontend-docker-image:latest
docker tag backend-docker-image:latest andyalemonta/backend-docker-image:latest
```

6.3 Pubblicazione su Docker Hub

Successivamente, le immagini sono state pubblicate su **Docker Hub** per consentirne l'accesso da qualsiasi ambiente. Questo passaggio garantisce che le immagini siano disponibili globalmente e possano essere utilizzate per distribuire i servizi su altri sistemi.

Sono di seguito riportati gli output della console durante il caricamento delle immagini.

```
andyalemonta@osteoarthritis-portal-vm:~$ docker push andyalemonta/frontend-docker-image:latest
The push refers to repository [docker.io/andyalemonta/frontend-docker-image]
b9dd38c96411: Pushed
22ad928f4d5d: Pushed
5ad457866708: Mounted from andyalemonta/frontend-image
bdaf0359a149: Mounted from andyalemonta/frontend-image
a11ed1ab3f37: Mounted from andyalemonta/frontend-image
0d5f5a015e5d: Mounted from andyalemonta/frontend-image
3c777d951de2: Mounted from andyalemonta/frontend-image
f8a91dd5fc84: Mounted from andyalemonta/frontend-image
cb81227abde5: Mounted from andyalemonta/frontend-image
e01a454893a9: Mounted from andyalemonta/frontend-image
c45660adde37: Mounted from andyalemonta/frontend-image
fe0fb3ab4a0f: Mounted from andyalemonta/frontend-image
f1186e5061f2: Mounted from andyalemonta/frontend-image
b2dba7477754: Mounted from andyalemonta/frontend-image
latest: digest: sha256:7b337ab36d047a2cbd06c33ca08f76f84b62cf8bd4c23b87b8ef9b295facf003 size: 3268
```

Figura 22: Pubblicazione dell'immagine Docker per il frontend

```
andyalemonta@osteoarthritis-portal-vm:~$ docker push andyalemonta/backend-docker-image:latest
The push refers to repository [docker.io/andyalemonta/backend-docker-image]
d5ba5741d6dd: Pushed
c845927df1f6: Pushed
80b997af8827: Pushed
2bf6136f8557: Pushed
fe5bbd4f8a42: Mounted from library/python
24f0c2413cd7: Mounted from library/python
8f9a13bfb118: Mounted from library/python
0aeeeb7c293d: Mounted from library/python
c81d4fdb67fc: Mounted from library/python
0e82d78b3ea1: Mounted from library/python
301c1bb42cc0: Mounted from library/python
latest: digest: sha256:5e8a3b20b87155bc987685b51893264d74a01825318dcaf961bfe1fe13ce07b3 size: 2637
```

Figura 23: Pubblicazione dell'immagine Docker per il backend

6.4 Deploy del sistema con Docker Compose

In Figura 24 è mostrato il processo di esecuzione del comando `docker-compose up --build -d` sulla VM, che avvia i container del sistema.

```
andyalemonta@osteoarthritis-portal-vm:~$ docker compose up --build -d
[+] Running 3/3
 ✓ Network andyalemonta_app-network  Created
 ✓ Container backend_container       Started
 ✓ Container frontend_container      Started
```

Figura 24: Deploy del sistema.

Una volta avviato il sistema, il sito è accessibile tramite il link http://34.121.167.35:8080/, dove l'IP corrisponde all'indirizzo pubblico della macchina virtuale che ospita l'applicazione.

7 Esempi di utilizzo

7.1 Registrazione e Login

Si suppone che un medico e un paziente vogliano utilizzare il sistema per caricare e analizzare radiografie. Inizialmente, entrambi visualizzano la schermata iniziale del sito, come mostrato in figura 25.

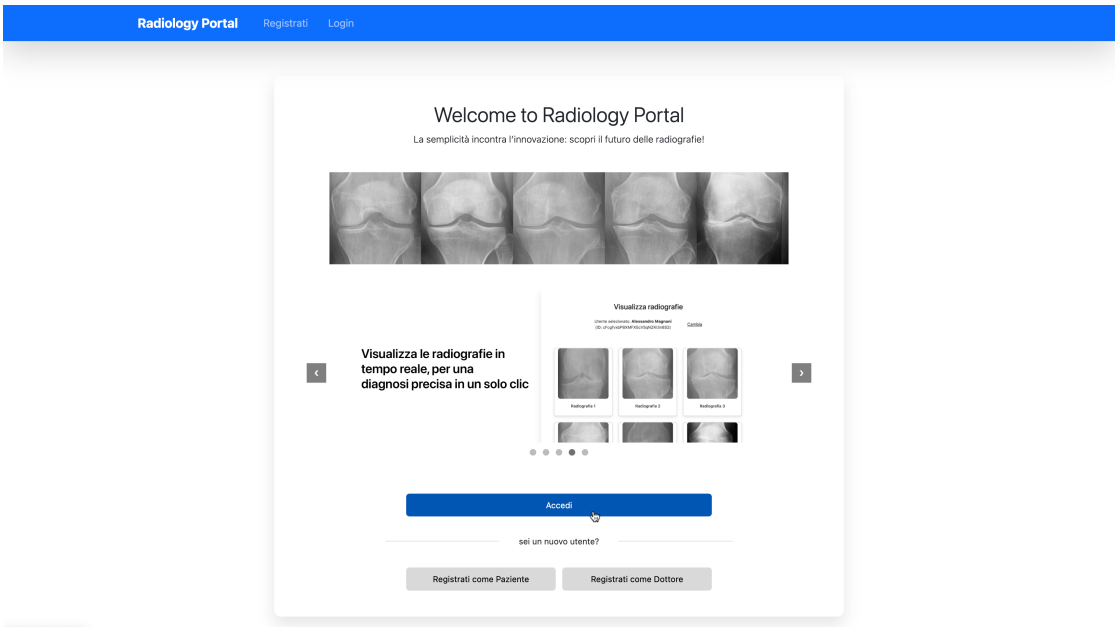

Figura 25: Schermata iniziale.

Entrambi devono registrarsi al sistema, seguendo un processo simile, con una piccola differenza nei campi richiesti durante la compilazione del modulo di registrazione.

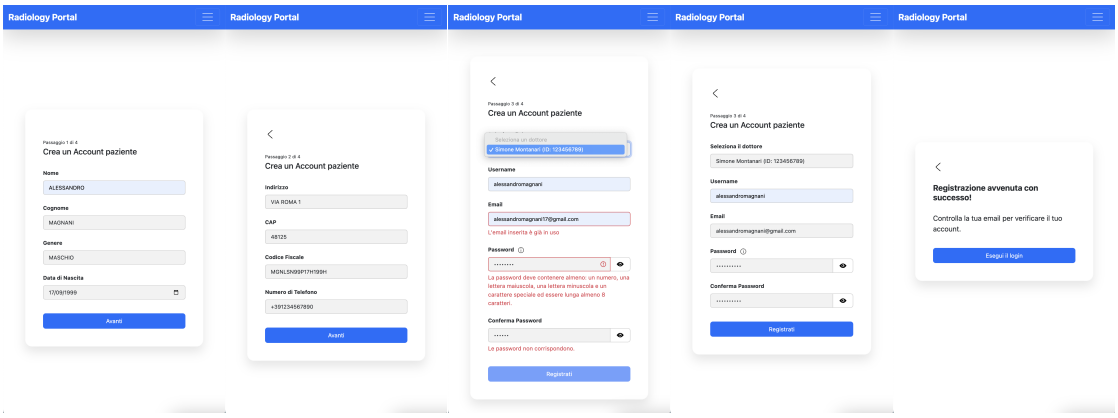

Figura 26: Schermate della registrazione.

Come si nota in figura 26, il processo di registrazione prevede i seguenti passaggi:

- **Accesso alla pagina di registrazione**.
- **Inserimento dei dati personali**.
  - **Medici:** inseriscono il proprio codice identificativo univoco.
  - **Pazienti:** selezionano il medico di riferimento.
- **Verifica dell'indirizzo email**.

Completata la registrazione, gli utenti ricevono un'email di verifica all'indirizzo fornito. Per attivare l'account, è necessario cliccare sul link presente nella mail, come illustrato in Figura 27.

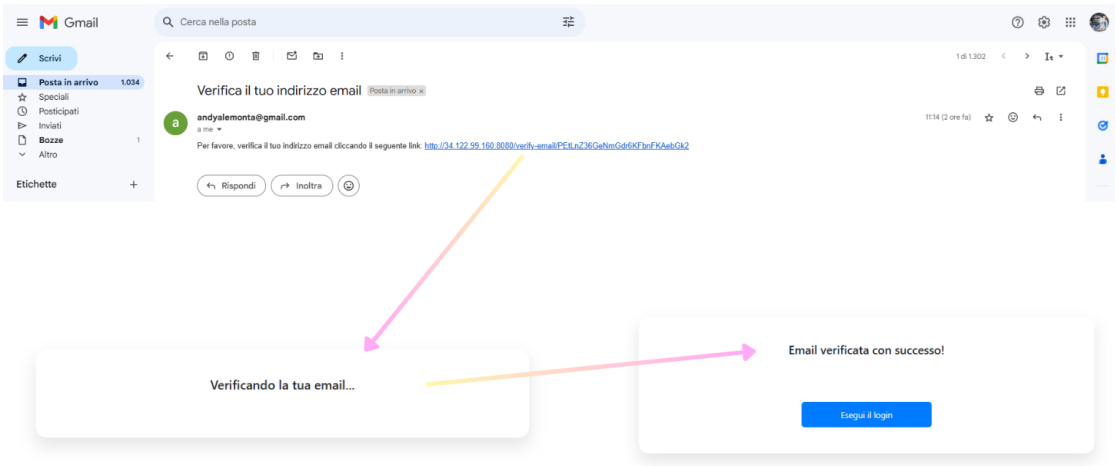

Figura 27: Processo di verifica dell'email.

Dopo aver verificato l'email tramite il link ricevuto, gli utenti possono accedere al sistema effettuando il login, come mostrato in figura 30.

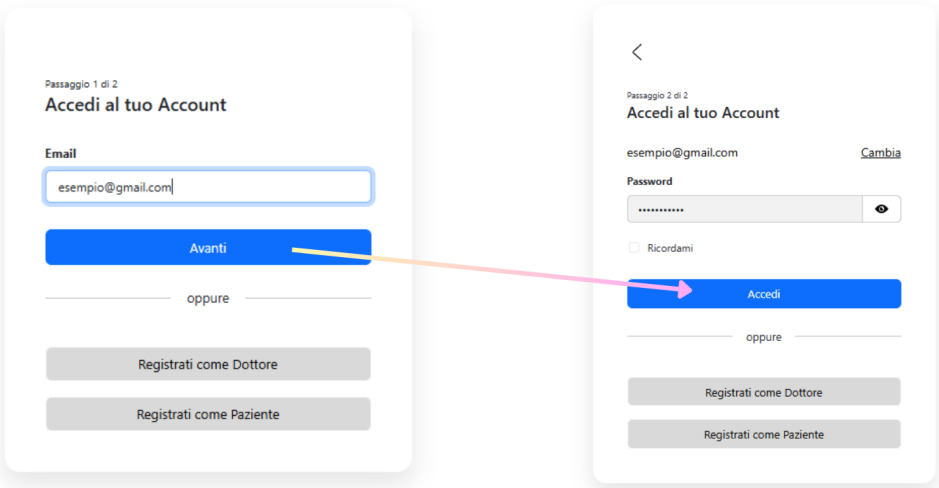

Figura 28: Processo di login.

Quando un utente tenta di effettuare il login senza aver precedentemente verificato il proprio indirizzo email, il sistema visualizza un messaggio di errore in rosso, indicando che la verifica dell'email è necessaria per completare il login.

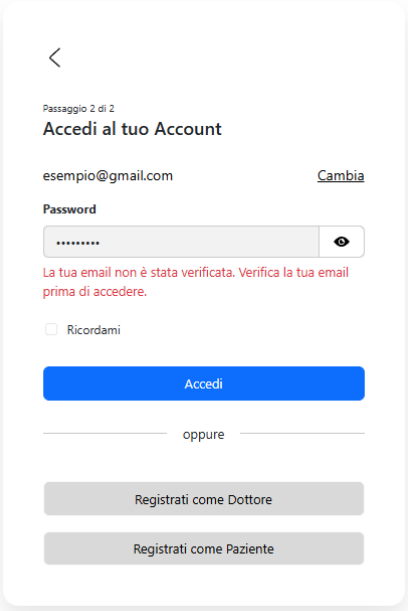

Figura 29: Email non verificata.

Nel caso in cui un utente inserisca una password errata ripetutamente, il sistema mostra un messaggio che indica i tentativi rimanenti. Quando i tentativi vengono esauriti, il sistema invia un'email per il reset della password.

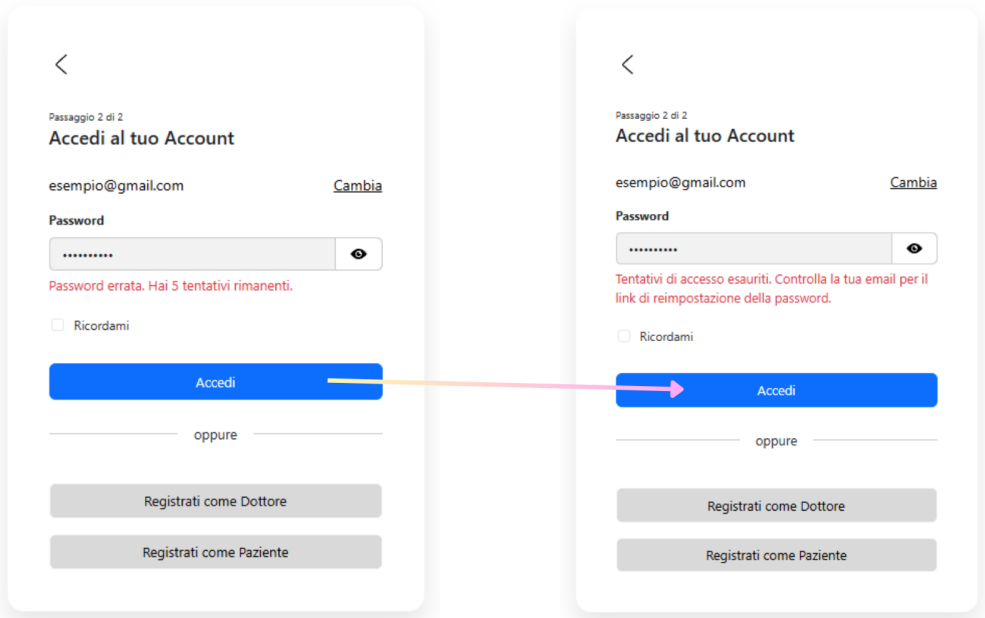

Figura 30: Errori nell'inserimento della password.

Una volta cliccato il link ricevuto via email, l'utente può reimpostare la propria password come mostrato in figura 34.

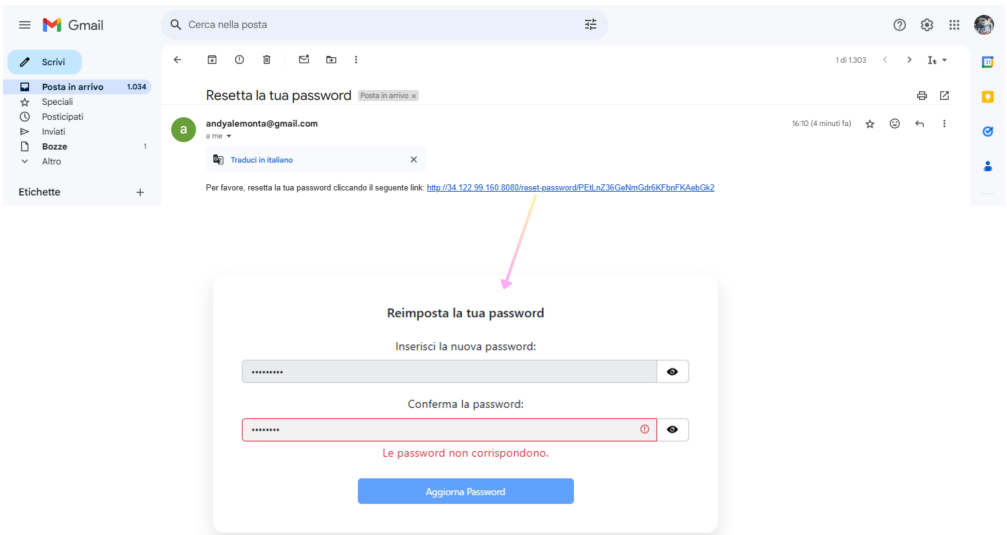

Figura 31: Reset della password.

7.2 Schermata di benvenuto e barra di navigazione

Una volta autenticati, gli utenti vengono reindirizzati alla schermata di benvenuto, mostrata in Figura 32.

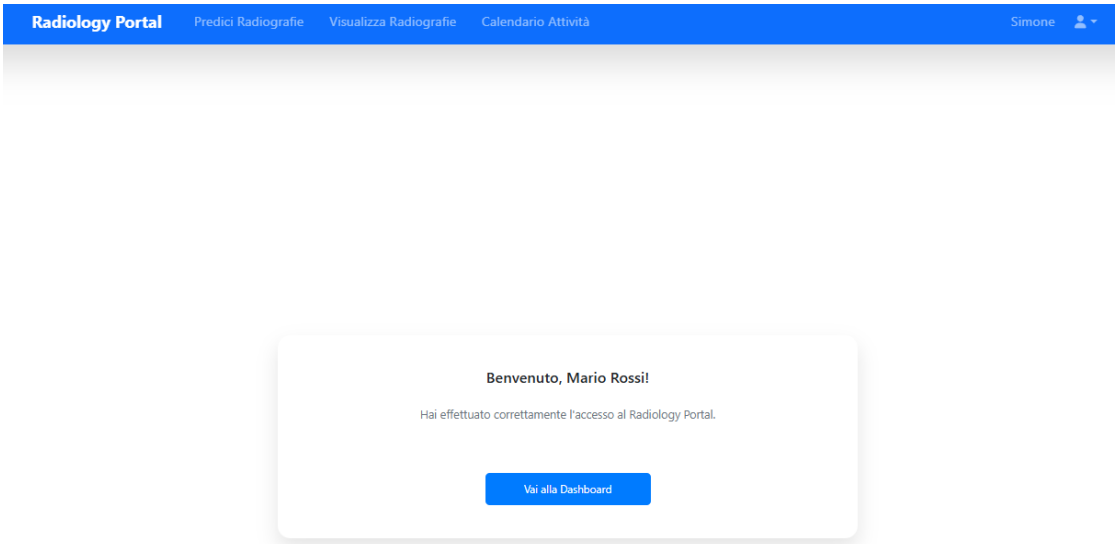

Figura 32: Schermata di benvenuto.

La figura 33 evidenzia le differenze nelle barre di navigazione di medici e pazienti.

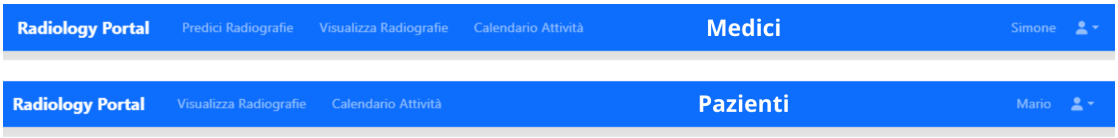

Figura 33: Differenze nelle barre di navigazione.

Questa schermata rappresenta il punto di accesso principale alle funzionalità del sistema come **Dashboard**, **Predizione dell'osteoartrite**, **Visualizzazione radiografie** e **Pianificazione di attività e notifiche**.

7.3 Dashboard

La dashboard rappresenta il punto di accesso centrale al sistema. I medici possono visualizzare un elenco completo delle informazioni anagrafiche e radiografie di tutti i pazienti a loro associati, mentre i pazienti possono consultare solo i propri dati.

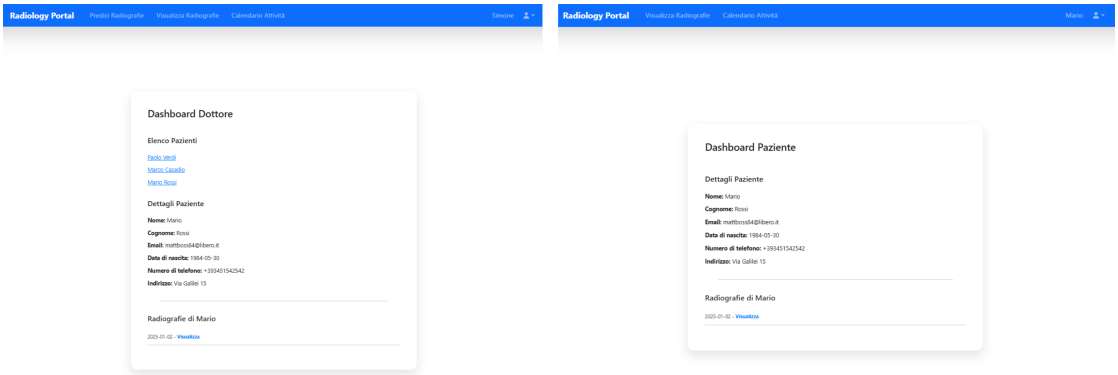

Figura 34: Dashboard per medici e pazienti.

7.4 Caricamento e predizione delle radiografie

In questa sezione, i medici possono caricare le radiografie dei pazienti e utilizzare il modello pre-addestrato (figura 35).

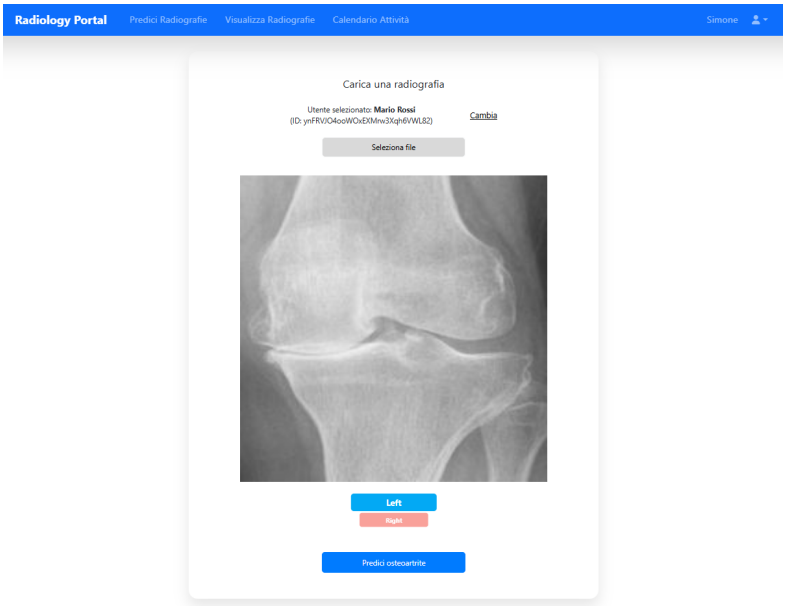

Figura 35: Caricamento di una radiografia.

Dopodichè, il sistema fornisce una predizione automatica, supportando il processo decisionale (figura 36).

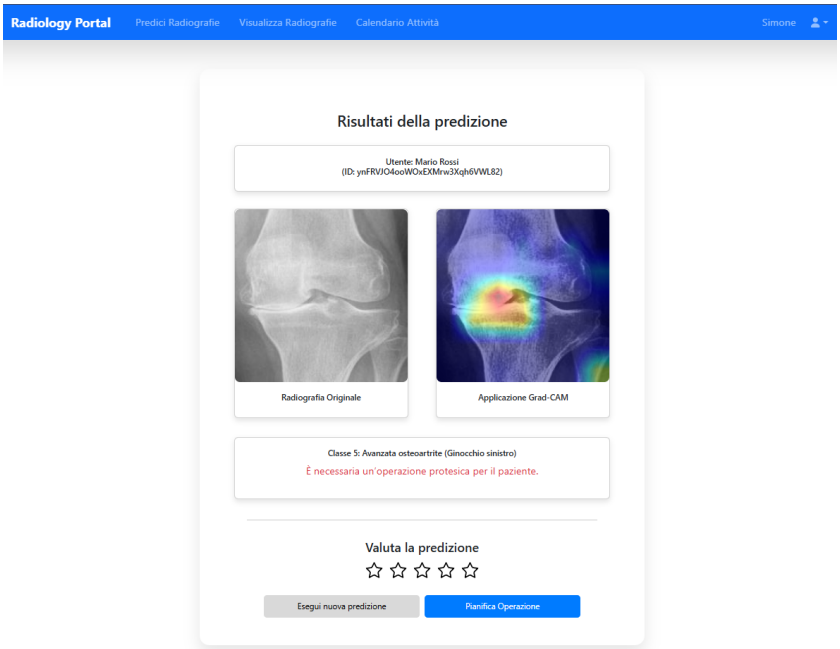

Figura 36: Predizione dell'osteoartrite.

7.5 Visualizzazione delle radiografie

I medici hanno la possibilità di visualizzare le radiografie di tutti i pazienti a loro associati con le relative predizioni (figura 37).

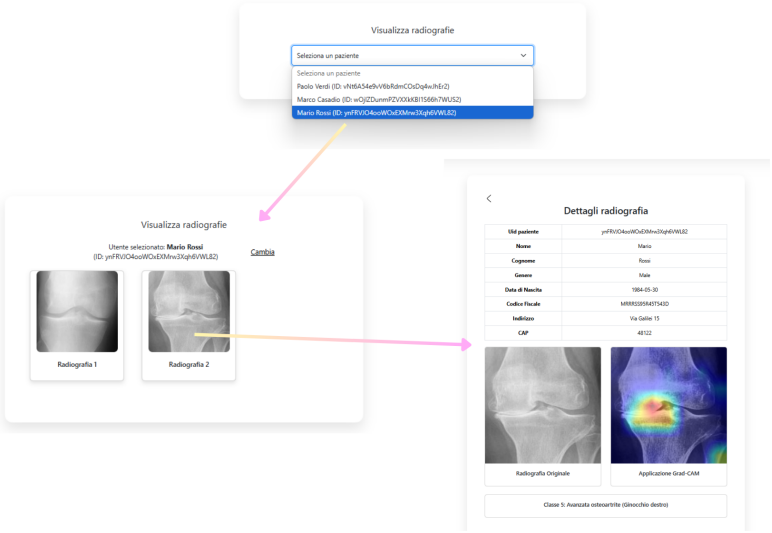

Figura 37: Visualizzazione radiografie medici.

I pazienti, invece, possono visualizzare solo le proprie informazioni (figura 38).

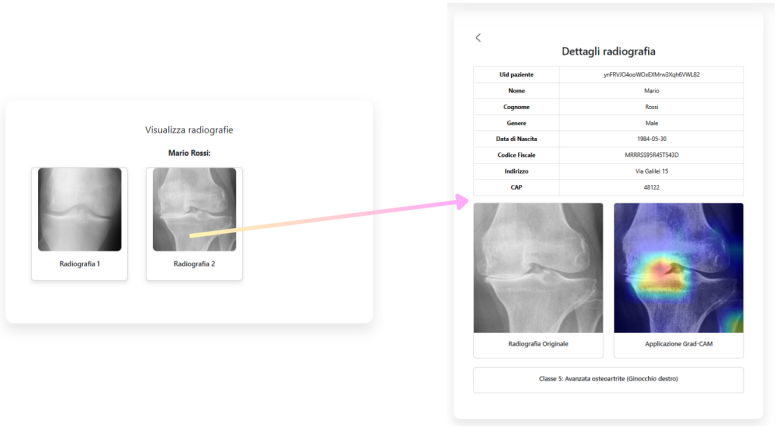

Figura 38: Visualizzazione radiografie pazienti.

7.6 Pianificazione di Attività e Notifiche

In questa sezione, medici e pazienti possono monitorare le attività, che includono le operazioni chirurgiche e il caricamento di radiografie. I pazienti possono visualizzare esclusivamente le informazioni che li riguardano, come mostrato in figura 39.

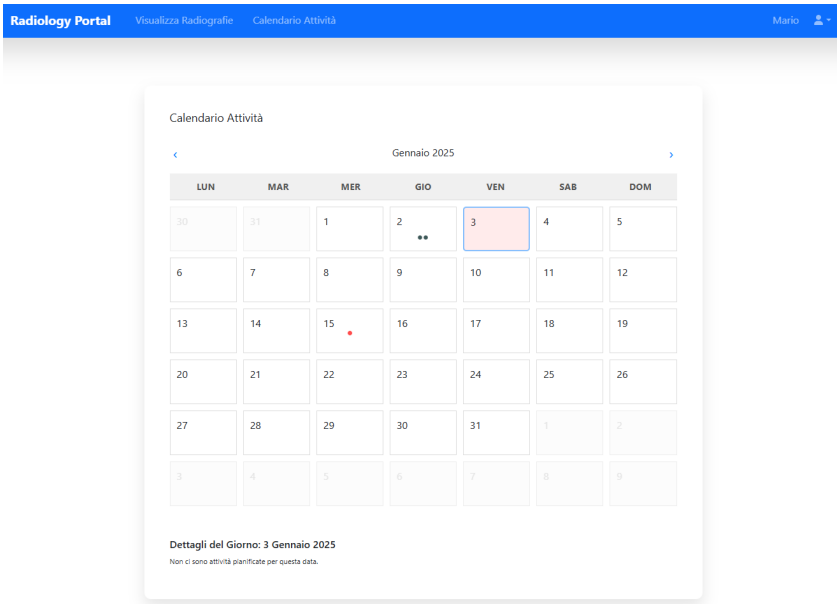

Figura 39: Vista del calendario lato pazienti.

I medici, invece, possono visualizzare le informazioni di tutti i pazienti associati (figura 40).

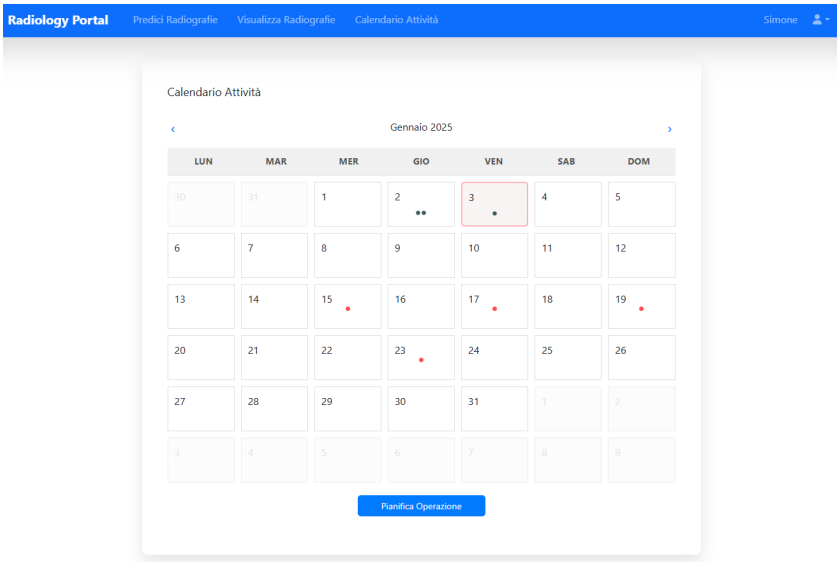

Figura 40: Vista del calendario lato medici.

Inoltre, possono pianificare un'operazione chirurgica inserendo tutti i dettagli relativi all'intervento. Una volta pianificata l'operazione, il sistema invia automaticamente una notifica al paziente interessato (figura 41).

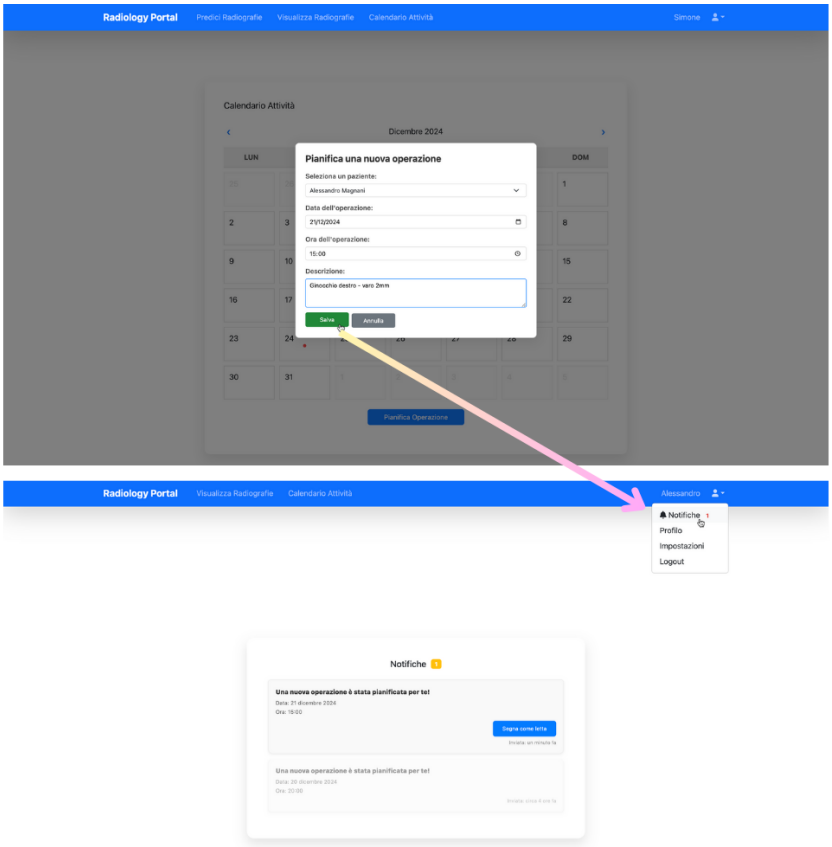

Figura 41: Processo di pianificazione di un'attività e ricezione notifica.

8 Conclusioni

Il progetto sviluppato si inserisce nel contesto della tesi che il team svolgerà in collaborazione con il primario di ortopedia dell'Ospedale Umberto I, con l'obiettivo di creare un sistema innovativo per la diagnosi dell'osteoartrite delle ginocchia attraverso l'analisi delle radiografie.

Durante lo sviluppo, i membri del team hanno acquisito e approfondito conoscenze significative nell'ambito delle applicazioni web, arricchendo la propria esperienza professionale.

La continua collaborazione tra i componenti del gruppo ha giocato un ruolo fondamentale nel raggiungimento degli obiettivi prefissati.

L'intero team ritiene che il progetto rappresenti un'importante opportunità di crescita e un'esperienza altamente formativa. Grazie a questo lavoro, il gruppo ha avuto l'opportunità di confrontarsi con aspetti complessi di progettazione e sviluppo, approfondendo tematiche relative all'architettura software, all'integrazione di sistemi complessi e alla gestione di dati sensibili in un contesto medico.

8.1 Sviluppi futuri

Il progetto offre diverse opportunità di sviluppi futuri. Tra queste, la possibilità di integrare la **WGAN-GP**, addestrata precedentemente da uno dei componenti del gruppo, per la generazione di immagini sintetiche con l'obiettivo di ampliare ulteriormente il dataset. Un'altra possibilità è quella di estendere il sistema ad altre patologie ortopediche, aumentando così l'applicabilità clinica.

Infine, un aspetto di grande importanza è la protezione dei dati sensibili, da perseguire attraverso tecniche avanzate di crittografia e gestione sicura delle informazioni. Questo permetterebbe di rispettare pienamente le normative vigenti, come il GDPR, e di garantire l'affidabilità del sistema per un utilizzo pratico in contesti clinici.

8.2 Cosa è stato appreso

Alessandro Magnani Questa esperienza è stata davvero arricchente sotto molti punti di vista. Ho avuto modo di ampliare le mie competenze con Vue.js e Flask, ma soprattutto ho imparato a muovermi meglio in un contesto più complesso grazie alle tecnologie di Google Cloud, come Firestore e Cloud Storage. Anche se non ho seguito direttamente tutto, lavorare a stretto contatto con i miei colleghi mi ha permesso di approfondire strumenti come Docker, Google Compute Engine, e concetti legati a Vertex AI e ai Job Scheduler. Ho anche avuto modo di utilizzare Axios, che non conoscevo, e di apprezzarne la semplicità nell'integrazione tra frontend e backend. Ho anche sperimentato approcci a me nuovi come le euristiche di Nielsen e il cognitive walkthrough. Nel complesso ho considerato questa esperienza molto utile e formativa, sono soddisfatto del lavoro e dei risultati ottenuti.

Simone Montanari Questo progetto è stato molto formativo, consentendomi di approfondire tecnologie che avevo solo sfiorato in passato. Il lavoro di squadra mi ha permesso di familiarizzare con strumenti utilizzati dai miei colleghi, ampliando così le mie competenze. Ho trovato il progetto estremamente interessante e spero che possa proseguire con successo in futuro.

Andrea Matteucci Questa esperienza, prima di tutto, mi ha permesso di migliorare la collaborazione in team: grazie al confronto costante con i miei colleghi, ho affinato le mie capacità di problem-solving e ho compreso l'importanza di gestire un'organizzazione condivisa. Parallelamente, ho avuto modo di esplorare più a fondo l'ecosistema Google Cloud, ad esempio su Cloud Storage e Vertex AI, e di capire meglio come gestire la

comunicazione tra front-end (Vue.js) e back-end (sfruttando anche Flask). Inoltre, ho imparato concetti rilevanti legati alla user experience, al fine di soddisfare realmente le esigenze degli utenti. Nel complesso, ritengo che questa attività abbia rappresentato un passo in avanti significativo per il mio percorso di apprendimento.

Note stilistiche

- Il **grassetto** è stato utilizzato per evidenziare aspetti importanti.
- Il *corsivo* è stato utilizzato per termini tecnici o parole inglesi non di uso comune nella lingua italiana, così come per espressioni o frasi di particolare enfasi o rilevanza.
- Il `monospace` è stato utilizzato per rappresentare aspetti relativi al codice o all'implementazione.

Riferimenti bibliografici

[1] Axios Team. Axios documentation, 2025. Accessed: January 2025. URL: `https://axios-http.com/docs/intro`.

[2] Docker Inc. Docker documentation - get started with docker overview, 2025. Accessed: January 2025. URL: `https://docs.docker.com/get-started/docker-overview/`.

[3] Flask Project Authors. Flask documentation, 2025. Accessed: January 2025. URL: `https://flask.palletsprojects.com/en/latest/`.

[4] Google Cloud Team. Google compute engine documentation, 2025. Accessed: January 2025. URL: `https://cloud.google.com/compute/docs`.

[5] Google Cloud Team. Google firestore and cloud storage documentation, 2025. Accessed: January 2025. URL: `https://cloud.google.com/firestore/docs`.

[6] Mozilla Developer Network (MDN). Cross-origin resource sharing (cors), 2025. Accessed: January 2025. URL: `https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS`.

[7] Node.js Foundation. Node.js documentation, 2025. Accessed: January 2025. URL: `https://nodejs.org/en/docs/`.

[8] Vue.js Team. Vue.js documentation, 2025. Accessed: January 2025. URL: `https://vuejs.org/guide/introduction.html`.